Chemnitz University of Technology

Department of Electrical Engineering and
Information Technology

CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Diploma Thesis

Finding Substructures of Molecules for Predicting Biological Activity

Heiko Hofer
born the 19th January 1978 in Zweibrücken

6th February 2003

Supervised by

Prof. Peter Protzel
Chair of Process Automation
Chemnitz University of Technology

Dr. Michael R. Berthold
Director of Data Analysis
Tripos Inc.

Contents

1 Introduction

1.1 Motivation

The main task of drug discovery is to find novel bioactive molecules. Bioactive molecules are for instance compounds that protect human cells against a virus. Conventionally drug candidates are synthesized and their biological activity is determined by running experiments (screening). The number of possible drug candidates is very large. Since screening tests are time consuming and expensive, it is a major challenge in drug discovery to select promising candidates.

Knowledge about the disease itself or about biological mechanisms can be used to choose drug candidates. Another way is to take a database of known active compounds and to find new molecules that are similar to those. It is expected that these new compounds are also active. They are slightly different to the existing compounds and there is the hope that they are more effective or that they have less side effects.

Similarity means in this case that similar compounds have the same behavior, that they for instance inhibit a particular disease. We base the similarity measure on the two dimensional structure of the compounds and we expect one or more active parts (fragments) in a biological active molecule. Those parts might be the key structures for the activity of the compound. If an unknown molecule contains the same active parts it is declared as "active". If it does not contain the active parts it is declared as inactive. It is not clear that this similarity measure leads to a similarity in behavior. Biologist and chemists have to check always if this similarity measure makes sense for a particular case.

Assigning an unknown molecule to the class of active or inactive molecules using an inter-molecular similarity measure is well known as *virtual screening*. Virtual screening can be used to reduce the number of drug candidates. Only those candidates that are expected to be active are then used in screening tests.

The main topic of this thesis is to find the active parts of active compounds based on a database of known molecules that are represented by their two dimensional structure. The database contains in general active and inactive compounds. Fragments are considered to be active parts if they are frequent in the active compounds and rare in the inactive compounds. Those fragments are called active fragments or discriminative fragments because they can be used to discriminate between the active and inactive compounds.

We used until now the biological activity to motivate the topic of this thesis.

But the approach is more general. It finds discriminative fragments between two classes of molecules. The reason for assigning a molecule to one class does not have to be its biological activity. Other applications are possible. For instance in the case of synthesizing molecules, the first class are the molecules that could not be synthesized and the second class are those that could be synthesized. In this case discriminative fragments could represent parts of molecules that inhibit the synthesis.

1.2 Related Work

Recently an approach was presented by Kramer, Raedt, and Helma (2001) that finds linear frequent fragments using an algorithm similar to the Apriori algorithm of Agrawal, Imielienski, and Swami (1993) for mining association rules. The restriction to linear chains of atoms is limiting in many real-world applications, since substructures of interest often contain rings or branching points.

An approach that finds arbitrary connected fragments has been presented by Borgelt and Berthold (2002). They use a sophisticated algorithm to generate fragments. It is a bottom up fragment generation method which takes a fragment for generating the next bigger ones. They have tested their approach on a dataset of the national cancer institute [NCI-HIV]. This work extends Kramers algorithm by considering arbitrary fragments instead of only chains.

The work of Kuramochi and Karipys (2002) is also an approach that finds all connected fragments that occur frequently over the entire set of compounds. The major difference to the previous approaches is the candidate (fragment) generation algorithm. They also use a bottom up method where the fragments at one level are used to create the next bigger ones by joining them. They use further graph theoretical methods like canonical labeling to reduce the number of expensive graph and subgraph isomorphism computations.

1.3 Aims of the Thesis

We will present an algorithm based on the work of Borgelt and Berthold (2002). Their approach itself is based on the Eclat algorithm by Zaki, Pathasarathy, Ogihara, and Li (1997) which is a method for mining association rules. An introduction to the association rule mining task and in particular the Eclat algorithm is given in Section 2.1.

The first aim of this thesis is to describe the underlying algorithm from a graph theoretical point of view. The description and the used definitions are presented in Section 2.2. There are further some examples that make the method more understandable.

A general problem for fragment mining methods are molecules with rings, because they "contain" a lot of different fragments. A database with many rings increases therefore the size of the problem significantly. For chemists and biologists rings are

one unit. They are usually not interested in fragments that contain only a part of a ring. The second part (Chapter 3) introduces a method that treats rings as special substructures and reduces the problem size by several orders of magnitude.

The third topic of the diploma thesis is the effective representation of the solution space. The solution space is the set of fragments that fulfill all constraints. An effective representation of the solution space needs only a minimum amount of fragments for reconstructing the whole solution space. We describe in Chapter 4 how the version space of Mitchel (1997) can be used for this issue. We introduce the reduced solution space which contains the fragments that are needed to determine the support of the fragments in the solution space. The support of a fragment are its frequencies in the classes of the database.

The fourth topic regards the similarity measure of fragments. All methods presented in the previous section consider fragments as different if they are structural different. But for chemists and biologist the similarity of fragments depends on the chemical context. They consider for instance fragments that differ in some atoms as equal, if the difference does not change the behavior of the fragment. Chapter 5 introduces a similarity measure that takes into account expert knowledge about structures that behave similar.

The presented approach is implemented in the object oriented language Java. Chapter 6 contains some interesting implementation details and the description of the graphical user interface.

The main topic of this thesis is to find discriminative fragments as described in Section 1.1. This problem is solved by a search algorithm that determines the frequency of a fragment in the active and in the inactive compounds. If a fragment is frequent in the active compounds and infrequent in the inactive compounds it will be called discriminative.

The algorithm that creates the fragments from a database of molecules is called candidate generation algorithm. The candidate generation algorithm is the main issue in the task of mining discriminative fragments. Our solution for the fragment generation and the task of mining frequent fragments is described in the next two chapters. Chapter 4 extends the method to the main topic of mining discriminative fragments.

2 Previous Work

Finding substructures for predicting biological activity is the main topic of this thesis. The prediction of biological activity is related to the classification of a molecule to the class of biological active molecules or to the class of biological inactive molecules.

In the introduction we noted that discriminative fragments are substructures that can be used as classification models. This approach uses a database that is split in the class of active molecules and in the class of inactive molecules.

Before we regard the task of mining discriminative fragments we will consider that the set of inactive molecules is empty. This is equivalent to the task of mining frequent fragments.

There is a related problem in the association rule mining task (Agrawal et al., 1993). The first step for mining association rules is to find frequent subsets in a database that consists of sets. In the next sections we will describe the Eclat algorithm by Zaki that solves this problem.

We will show afterwards that a similar approach can be used for mining frequent fragments.

2.1 Association rule mining

The Eclat (**E**quivalence **CLA**ss **T**ransformation) algorithm has been introduced by Zaki et al. (1997). It is an efficient way to solve the problem of finding frequent subsets in a database. Finding frequent subsets is the first step for mining association rules (Agrawal et al., 1993).

The Eclat algorithm takes a database $D = \{d_1, d_2, \ldots d_m\}$ which is a set of m sets. It exists a set S_0 with $S_0 = \{a, b, \ldots\}$ and $|S_0| = n$ where all d_i are subsets of S_0 with $i, \ldots, m$. Each element of the database is therefore a subset of the power set of S_0.

A very nice representation of the power set $\mathcal{P}(S_0)$ is a lattice as shown in Figure 2.1. It contains all possible combinations of the elements in S_0. The empty set is at the bottom and at the next level are the elements of S_0. They are then combined to pairs, and further combinations lead to the whole set S_0 at the top.

The task of finding frequent subsets can be done by traversing the power set lattice in any way and determining for each node the frequency in the database D. The measure for the frequency is often called support (Agrawal et al., 1993), (Zaki

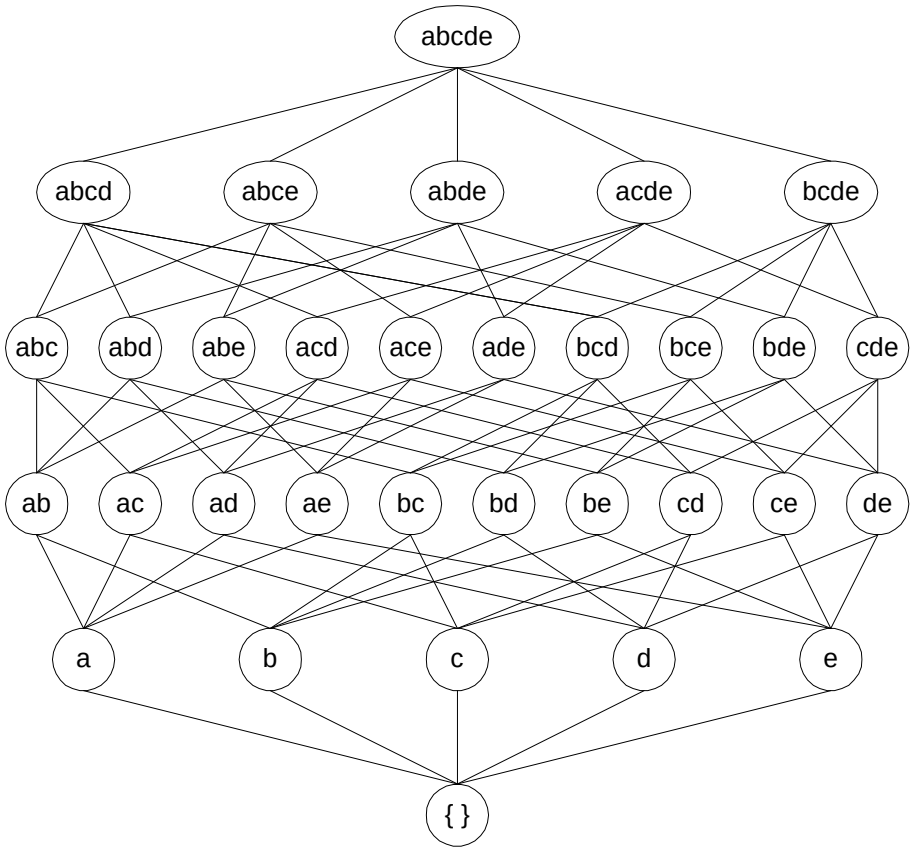


Figure 2.1: The lattice that displays the power set $\mathcal{P}(S_0)$ with $|S_0| = 5$

et al., 1997) and is defined by

$$\mathit{freq}(x, D) = \frac{|\{d_i \mid x \subseteq d_i,\ i = 1 \ldots m\}|}{|D|} \quad \text{with} \quad x \in \mathcal{P}(S_0),$$

which is the relative frequency of $x \in \mathcal{P}(S_0)$ in the database D. The set $\{d_i \mid x \subseteq d_i,\ i = 1 \ldots m\}$ is the set of elements in the database that contain x. It is called the tid-list[1] of x.

The major difference between the Apriori algorithm (Agrawal et al., 1993) and the Eclat algorithm (Zaki et al., 1997) is how to traverse the power set lattice and how to determine the frequency of a subset in the database. The Apriori approach performs a breadth-first, bottom-up search through the power set lattice. It creates the next level by joining the frequent subsets at a level. The supersets of a rare subset are therefore not created which is allowed, because *all supersets of a rare set $x \in \mathcal{P}(S_0)$ are rare.* The frequency can only decrease from one level to the

[1] tid means transaction identification. Agrawal introduced the term *transaction* for an element of the database.

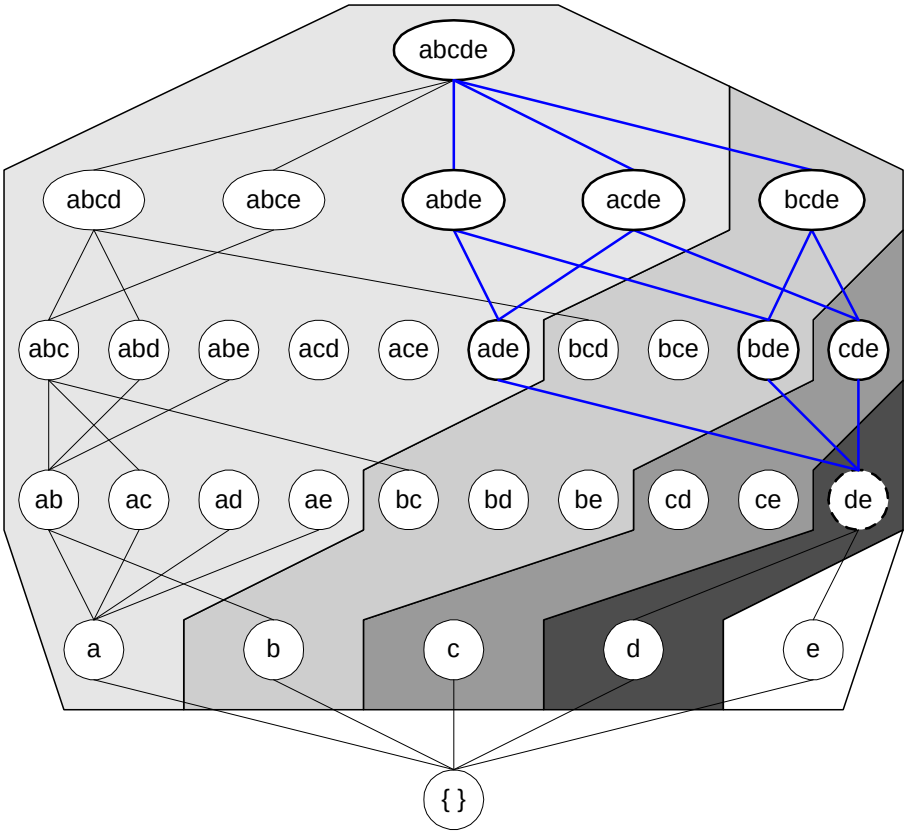


Figure 2.2: The equivalence classes that are use by the Eclat algorithm

next higher level in the lattice. Apriori determines the frequency of a subset x by counting the elements in the database D that contain x.

The Eclat algorithm in contrast, uses a depth first search for traversing the power set lattice. It first splits the power set lattice in sublattices using a lexicographical order as shown in Figure 2.2. Each of those sublattices can be processed on their own (Zaki, 1998). The major benefit of dividing the power set lattice is that not all sublattices are in main memory at the same time . However, in some cases, a sublattice may still be too large to be solved in main memory. In this scenario, recursive lattice decomposition will be applied until the biggest sublattice is small enough.

The nodes in an isolated sublattice can either be traversed in bottom up, breadth first or in a bottom up, depth first manner. But it is important to solve the sublattices in a reverse lexicographical order. For the example in Figure 2.2 this means to start with sublattice e, then d and so on. This order has the advantage of using the same pruning strategy as Apriori. If e. g. node *de* is not frequent than all nodes that are connected with this node are not frequent (highlighted with bold lines in Figure 2.2), because node *de* is a subset of these nodes. Since the sublattices

are traversed in a reverse lexicographical order all those nodes are processed after *de*. Therefore using this order it is possible to reduce the nodes that have to be processed by using information from the processing of the preceding sublattices.

The Eclat approach determines the frequency of a subset in a different way than the Apriori algorithm. It remembers the tid-lists of each node. The tid-list of a node x results from joining the tid-lists of the nodes that are subsets of x in the next lower level. The frequency of a subset results from the number of elements in the tid-lists divided by the number of elements in the database.

2.2 The molecular approach

Let us consider the task of mining fragments in a molecular database in this section. We use a database of molecules where the molecules are given in their two dimensional representation. Figure 2.3 displays an example, it is a molecule that belongs to the group of styryls. Its chemical name is 5,6-dimethyl-2-styryl-4(3H)-pyrimidinone.

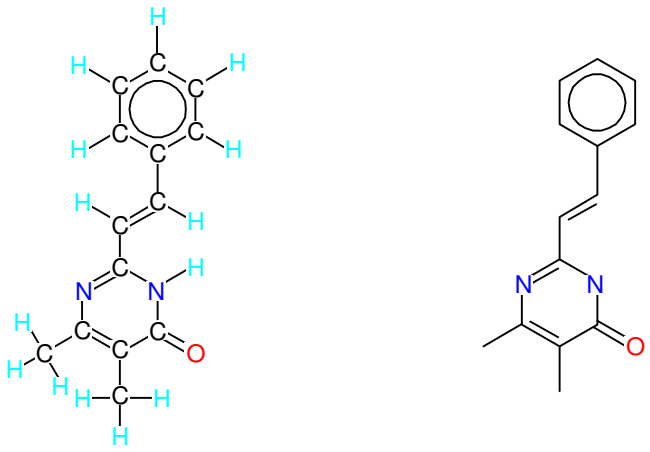


Figure 2.3: Example of a molecule.

At the left the full molecule is displayed with all hydrogen atoms and with explicitly drawn carbon atoms. Since carbon atoms are frequent in organic chemistry they are usually not explicitly drawn as is shown on the right of Figure 2.3. This makes the molecule more readable. The hydrogen atoms are also not shown at the right. The number of hydrogens at one atom can be calculated from the atom's valency number and the number of connected valencies. The molecule at the right therefore displays the same information, but it is a smaller graph, which decreases the size of the problem.

The molecule in Figure 2.3 for us consists of 17 atoms and 18 bonds. There are one oxygen atom, two nitrogen atoms and 15 carbon atoms. The circle in the upper ring indicates that the six bonds in this ring are aromatic.

When we imagine the substructures of the example, it is clear that we can arrange them in a lattice, too. At the bottom are the single atoms O, N and C. The next level are all pairs that can be found in the example and at the top is the molecule itself.

When we want to adopt methods of association rule mining we have to make sure that assumptions that hold for sets hold also for molecules. The first problem is that tid-list intersection does not work with fragments of molecules, because when $N = C$ and $C = O$ are fragments of a molecule, $N = C = O$ is not necessarily part of the same molecule. However, the second assumption that frequencies of nodes in the lattice decrease at higher levels hold for the lattice of fragments, too. The fragment $N = C$ has at least a frequency greater or equal than the frequency of the fragment $N = C = O$. Or in other words the set of molecules that contain the fragment $N = C = O$ is a subset of the molecules that contain the fragment $N = C$.

Another problem is that it is much easier to build the power set lattice than the lattice of fragments. Borgelt and Berthold (2002) have solved those problems and they have introduced an algorithm that uses a depth first search to traverse the lattice of fragments.

Before we have a closer look at their algorithm we will define the problem in graph theoretical terms. One possible model for the two dimensional representation of a molecule is a labeled graph. A labeled graph G consists of a set of vertices V and a set of edges $E \subseteq [V]^2$; thus, the edges are two-element subsets of V and if the set $\{v_i, v_j\}$ is an element of E then the vertex v_i and the vertex v_j are connected. A function $l_V : V \rightarrow L_V$ assigns a label $l \in L_V$ to each vertex. In molecular graphs, the vertices correspond to the atoms of the molecule and the label of a vertex is the atomic number. Analogously, a function $l_E : E \rightarrow L_E$ assigns a label $l \in L_E$ to each edge and the edges correspond to the bonds of the molecule. In the following we will denote a labeled graph by $G = (V, E, l_V, L_V, l_E, L_E)$.

A fragment of a molecule can also be modeled by a labeled Graph. The fragment is a subgraph G' of a Graph G denoted by $G' \subseteq G$, whereas the conditions $V' \subseteq V$, $E' \subseteq E$, $l'_V(v'_i) = l_V(v'_i)$ and $l'_E(e'_i) = l_E(e'_i)$ for all i are fulfilled. The Graph G is called the supergraph of G'. In the case of molecular graphs we will only consider connected subgraphs. Connected subgraphs are subgraphs that consist of one part only.

Using the definitions we can formulate the problem of finding frequent fragments as follows. Given is a database which is a set of n labeled graphs. The set of connected subgraphs are the pairwise not equal fragments of these graphs and we are interested in those fragments that are part of a minimum number of graphs in the database. They are called *frequent fragments*.

The problem formulation involves the set of connected subgraphs. Its size is called the size of the problem. It is not straightforward to estimate the number of fragments. For a database of $|V|$ vertices where each pair of vertices is connected

by an edge we can derive an upper bound of

$$N_{max} = \sum_{k=1}^{|V|} \left[\begin{pmatrix} |V| \\ k \end{pmatrix} \sum_{p=k-1}^{|E_k|} \begin{pmatrix} |E_k| \\ p \end{pmatrix} \right] \quad \text{with} \quad E_k = \frac{k(k-1)}{2}$$

fragments and for a database of $|V|$ vertices where no vertices are connected we find the lower bound to be a number of $N_{min} = |V|$ subgraphs.

If $|V| = 10$ the extreme values are $N_{min} = 10$ and $N_{max} \approx 3.6 * 10^{13}$. The truth for molecular graphs lies somewhere in between. Atoms in a molecular graph have few neighbors. They are therefore not highly connected. There exist four different kind of edges (single, double, triple and aromatic) and there are over 100 different kind of atoms but they are for the most part carbons in organic chemistry. The molecule in Figure 2.3 contains for instance only 1366 pairwise unique fragments.

We will use a database from the National Cancer Institute to present our results [NCI-HIV]. The molecules in this database are, on average, approximately four times bigger than the molecule in Figure 2.3. Since the number of subgraphs does not scale linearly and the databases contain thousands of molecules it is clear that we need a sophisticated method to solve a problem of this size.

The main part of the algorithm from Borgelt and Berthold is a method that creates the fragments of a molecular graph. This approach is described next and the algorithm itself is described in Section 2.3.

Bottom up fragment generation

Building all connected subgraphs of a labeled graph is not an easy task, because simple combination of the vertices will lead to a lot of equal or not connected subgraphs. An interesting idea to avoid these problems is to start from a connected subgraph (core), which can be the empty subgraph and iteratively add edges and vertices of the database to the core, so that only connected supergraphs are created. These graphs will be connected and each graph will be a subgraph of at least one graph in the database, because only edges and vertices of the database have been added. This significantly reduces the number of created subgraphs, in comparison to the breath first combination algorithm.

The algorithm for creating all subgraphs of a graph is described next. We start with some definitions followed by the definition of the complete algorithm. The algorithm is then validated on some key substructures of graphs.

We want to avoid that equal subgraphs are created. The equality of graphs is called isomorphism in graph theoretical terms. A labeled Graph $G = (V, E, l_V, L_V, l_E, L_E)$ is *isomorphic* to the labeled Graph $G' = (V', E', l'_V, L'_V, l'_E, L'_E)$ if there exists a bijection $\varphi : V \to V'$ with $\{v_i, v_j\} \in E \Leftrightarrow \{\varphi(v_i), \varphi(v_j)\} \in E'$, $l_V(v_i) = l'_V(\varphi(v_i))$ and $l_E(\{v_i, v_j\}) = l'_E(\{\varphi(v_i), \varphi(v_j)\})$ for all $v_i, v_j \in V$.

We start from a core as mentioned above. The core is considered to be a labeled graph $G = (V, E, l_V, L_V, l_E, L_E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$. An extension of G is a supergraph $G_{ext} = (V_{ext}, E_{ext})$ of G with

$V_{ext} = \{v_1, v_2, \ldots, v_n, v_{n+1}, \ldots, v_{n\,ext}\}$ and $E_{ext} = \{e_1, e_2, \ldots, e_m, e_{m+1}, \ldots, e_{m\,ext}\}$. For now we do not consider extensions with more than one vertex. They are called *substructure extensions* and we will regard them in Chapter 3. Thus extensions with one vertex and one edge, and extensions with only one edge are allowed.

It is furthermore important that each graph in the searched set of subgraphs has a vertex $v_r \in V$ which is restricted extendable. Vertices v_i with $i < r$ are not extendable and the vertices v_i with $i > r$ are (unrestricted) extendable. The restriction of v_r depends on the neighbors of v_r that are unrestricted extendable. It is only allowed to add a new edge to v_r if the attachment to this extension is greater than to the extendable neighbors of v_r. The bijection $\psi : L_E \times L_V \to \mathcal{R}$ is called the attachment function and it defines an order for all edge-vertex pairs. Hence the restricted vertex v_r can only be extended by extensions with $\psi(extension) \geq \max\{\psi(neighbors)\}$.

Up to now we still miss a rule which defines the restricted vertex in a subgraph. The index r of the restricted vertex is determined in an iterative way. It is initialized for the core to zero. All vertices of the core are therefore unrestricted extendable. If a subgraph is extended by an edge-vertex pair, the restricted vertex will be the one that has been extended. Extensions that consist only of an edge must close rings. We explain in paragraph rings on page 12 that it is allowed to extend a subgraph at any vertex by an edge extension. There is no restriction in this case, but a subgraph that has been extended by one or more edges can not be extended by an edge-vertex pair.

The following examples will verify the rules by building the set of subgraphs for some key structures. The set $L_E = \{-, =\}$ will be used to label edges and the set $L_V = \{x, y, z\}$ for vertices. The order of attachment is:

$$-x \quad < \quad =x \quad < \quad -y \quad < \quad =y \quad < \quad -z \quad < \quad =z$$

Chains The first example is the chain in Figure 2.4. We start with the core. The

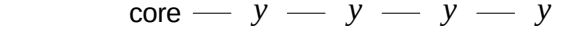


Figure 2.4: The database of the chain example.

core can only be extended by $-y$ which creates the extension core$-y$. It is allowed to extend the core because all vertices in the core are initialized as extendable. One vertex in the core of core $- y$ is set to be restricted extendable because $-y$ is an edge-vertex pair extension. However the only possible extension of core $- y$ is the subgraph core$-y-y$. This extension is also allowed because the vertex y in core$-y$ is unrestricted extendable. It is clear that the attachment rule is not needed in this example. Arbitrary extensions can be added to the chain. The tree of subgraphs is displayed in Figure 2.5, where the restricted extendable vertex is marked by a tilde and the extendable vertices are underlined.

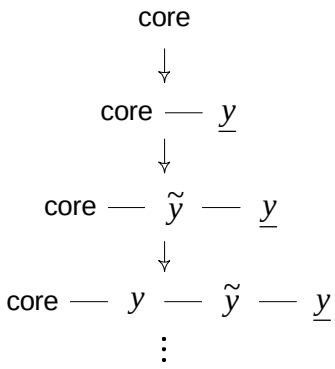


Figure 2.5: The tree of subgraphs for the chain example.

Intersections An intersection is a vertex with more than two neighbors. Figure 2.6 displays one graph that consists of a core and two chains leaving from an intersection. The beginning is the same as in the chain example. The subgraph core $-\underline{y}$

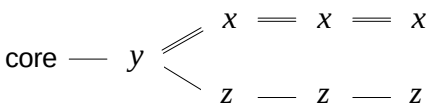


Figure 2.6: The database of the intersection example.

will be created with y as an extendable vertex. But this subgraph is extendable by the atoms x and z and both extensions will be created, because y is unrestricted extendable. The intersection point y is the restricted extendable vertex for both extensions core $-\tilde{y} = \underline{x}$ and core $-\tilde{y} - \underline{z}$. This is important for the generation of the next extensions. Figure 2.7 shows that the subgraph core $-\tilde{y} = \underline{x}$ has two extensions, but core $-\tilde{y} = \underline{z}$ has only one extension. The reason is that the restricted extendable vertex in subgraph core $-\tilde{y} - \underline{z}$ is not extended by $= x$ because the attachment of the extension $= x$ is smaller than the attachment of the neighbor $-z$. The advantage of the attachment rule is that no isomorphic subgraphs will be generated in case of intersection points. This is very important, because intersections are frequent in chemical graphs. Figure 2.7 displays the tree of subgraphs for the intersection example. The indices of vertices are sometimes displayed relative to the restricted extendable vertex which has the index r. This is for a better understanding in cases where more than one vertex is unrestricted extendable.

Rings Rings are the third basic substructures in a graph. Chains and intersections are graphs with n nodes and $n-1$ edges, called *trees* (see Diestel (1997) page 13). Those graphs are dividable into edge-vertex pairs when the core is also a tree. Graphs with rings have more edges and are thus not dividable into edge-vertex pairs, which is the main difference to the previous examples.

For a database with rings, the set of all subgraphs will consist of subgraphs

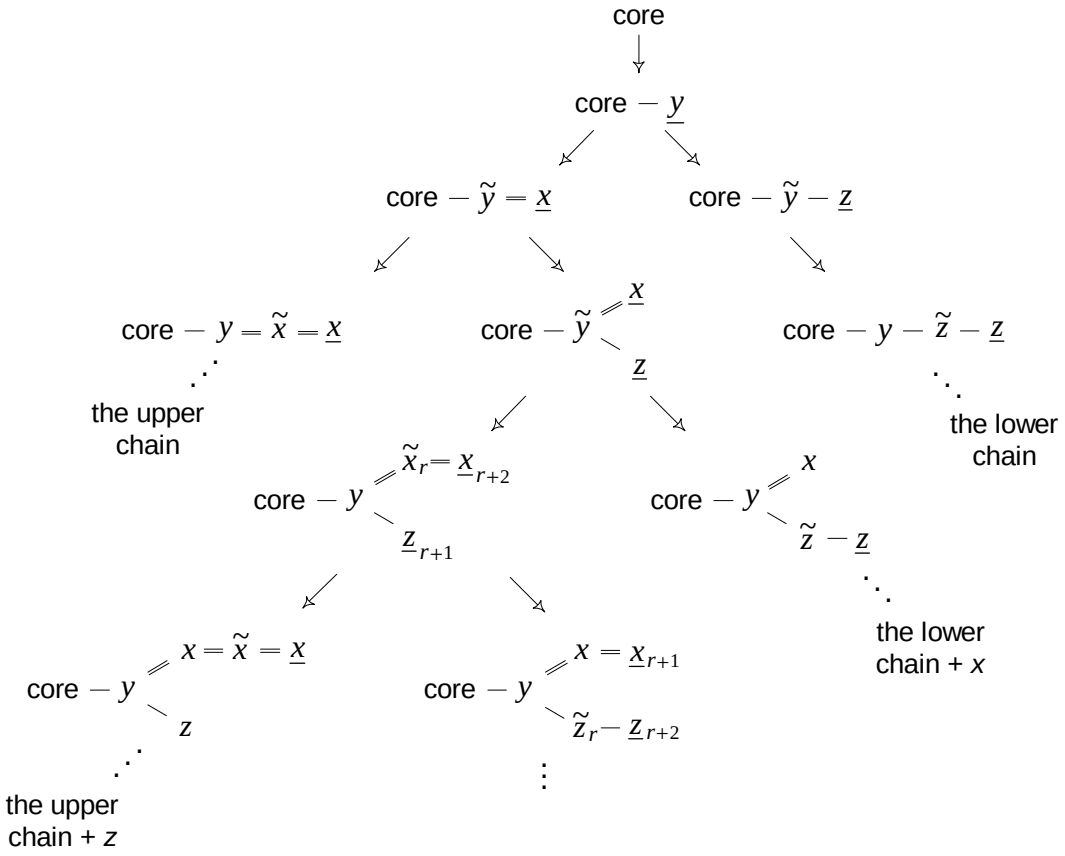


Figure 2.7: The tree of subgraphs for the intersection example.

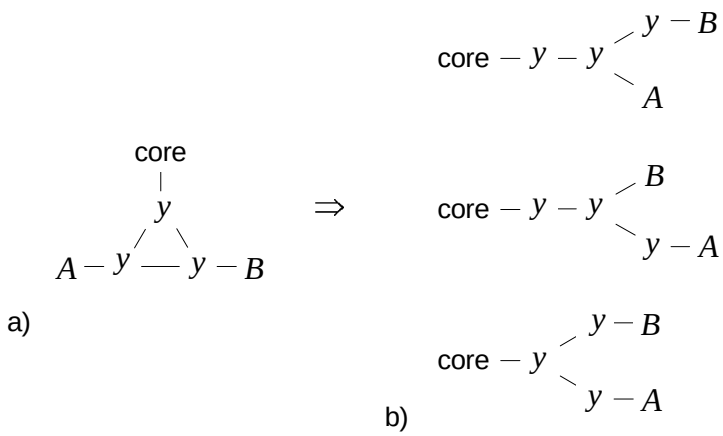


Figure 2.8: The database of the ring example and its decomposition into graphs without rings.

without rings and their counterparts with the same atoms but more edges that close rings. If our algorithm creates all subgraphs without rings it does also create all counterparts, because edge extensions can start from any atom of a subgraph. We will verify this statement for the graph in Figure 2.8 a), where A and B are arbitrary subgraphs. The graphs in Figure 2.8 b) have been built by removing one edge of the ring. They are called spanning trees of the graph, because they contain all vertices of the graph but they have no ring. If our algorithm creates all subgraphs of those graphs it will also create all subgraphs of the graph with the ring. Figure 2.9 displays the tree of subgraphs if extensions that consist only

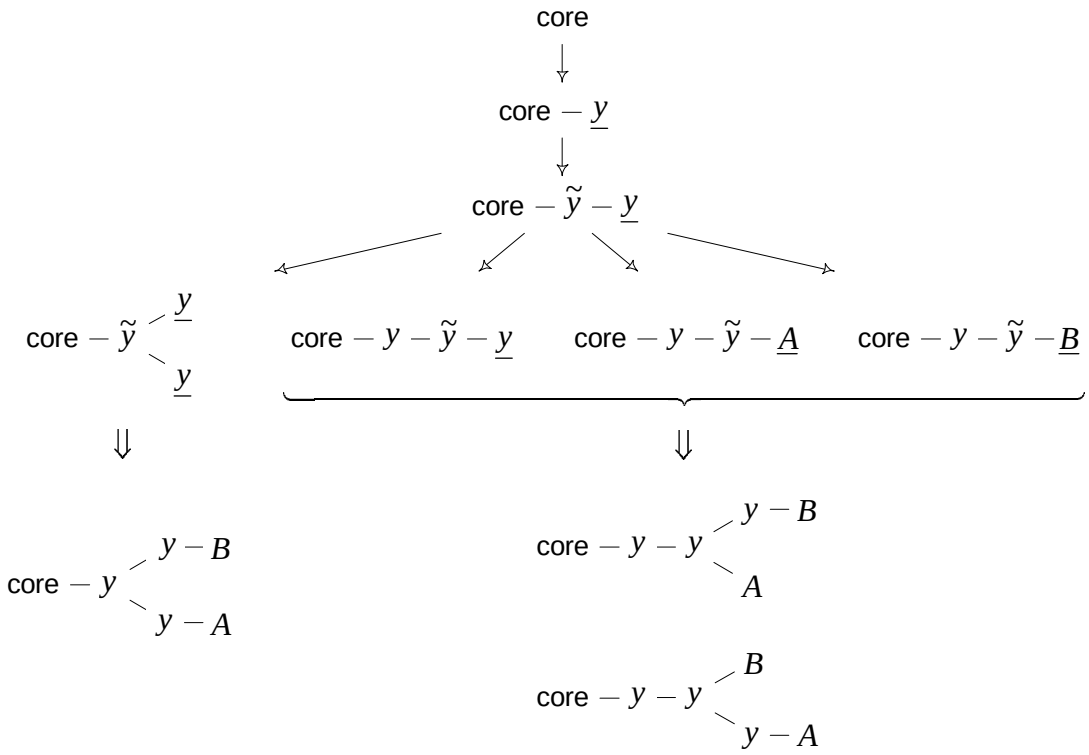


Figure 2.9: The tree of subgraphs for the ring example, if edge extensions are not allowed.

of one edge are not allowed. The left branch creates the substructures of the last spanning tree. The other spanning trees are created by the remaining branches at the right. It depends on the attachment of the first vertices in A and B comparing to $-y$ which branch will contain the whole spanning tree.

If the substructure B is just one vertex, the subtree that is displayed in Figure 2.10 will be in the left branch of the whole subgraph tree. Extending the root of the subtree by an edge extension closes the ring. Adding $-B$ to this subgraph would built the same subgraph that can be created by adding $-B$ to the root and adding the edge to the resulting subgraph. It is therefore not allowed to add an edge-vertex pair to a subgraph that was extended by an edge extension.

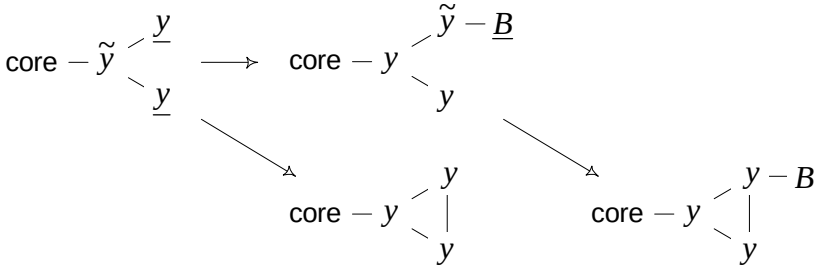


Figure 2.10: A subtree of the subgraph tree to the ring example.

The whole graph can be created from each spanning tree by adding an edge. Thus, it is created three times in different branches of the subgraph tree. There are other subgraphs like core $- y - y - A$ that are created twice, because they are part of two spanning trees. We can conclude that equal subgraphs will be created in different branches of the substructure tree in case of graphs that contain rings.

2.3 The Molecular Fragment Miner

The last section gives a detailed description of a method to create the subgraphs of a graph. The output of the algorithm is a tree that contains the subgraphs of a graph or of a set of graphs. Note that the algorithm does not create the lattice of fragments. The difference between the tree and the lattice is that some connections are missing but disregarding those connections allows us to design a more efficient algorithm.

The algorithm that is presented next is called MoFa (Molecular Fragment Miner) and it solves the problem of finding frequent fragments. MoFa takes a database of labeled graphs and a core which is the starting point for the search.

Embed core One feature of MoFa is the initialization with a core. Only those fragments that contain the core are considered by MoFa. A bigger core can therefore be used to restrict the search because less fragments contain this core. But it is also possible to do a full search when the core is equal to the empty graph.

If the core is the empty graph we first determine the frequency of different nodes in the database. When we consider the molecular graph in Figure 2.11 as the database, we would determine that there are one oxygen atom, two nitrogen atoms and 14 carbon atoms. The algorithm is then restarted with the least frequent atom. So in the example the oxygen is taken as the core. The fragments with at least one oxygen atom are processed in this run.

The next run is started with the nitrogen atom, the second least frequent atom, whereas only extensions are allowed that do not contain an oxygen atom. In this run all fragments are processed that contain at least one nitrogen atom but no oxygen atoms.

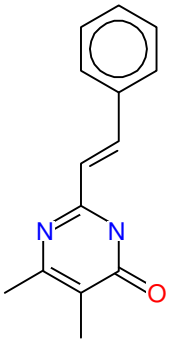


Figure 2.11: Molecular graph example.

The last run is started with a carbon atom. In this run only pure carbon fragments are regarded. This method is similar to the approach of the Eclat algorithm. The lattice of subgraphs will be partitioned in exactly the same way as displayed in Figure 2.2 on page 7.

In case the core is not empty, it is embedded in all graphs of the database. The result will be a list of locations of the core in the database. The locations can be found by using the core as a template. The first atom of the core is searched in the beginning. It is then extended by atoms and bonds of the core until the embeddings are isomorphic to the core. The list of embeddings themselves are pairwise identical. They represent one isomorphism of the core.

When we regard for instance the core N – C, we would find it three times in the molecular graph displayed in Figure 2.11. This list of core-embeddings is the starting point for further processing.

Extend core The next step is to extend the core-embeddings using the approach that is explained in the previous section. Since all vertices of a core are unrestricted extendable, all possible extensions are created. The algorithm tests for each embedding if any extensions are possible. The left embedding has three possible extensions. The carbon atom can be extended by a single bond to a carbon atom and by a double bond to another carbon atom. The nitrogen atom can be extended with a double bond to a carbon atom. The core embeddings at the right are both extendable in three different ways. All extensions of the core embeddings are listed in Figure 2.12.

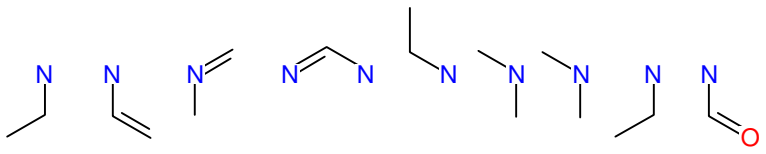


Figure 2.12: The extensions of the core N – C.

Sort extensions The extensions are sorted in lists that contain equal extensions. There are six different extensions of N − C. As mentioned before the equality of graphs is called isomorphism in graph theoretical terms. Fortin (1996) explains that it is expensive to verify the isomorphism of two graphs. Since we regard all spanning trees originated by the core, we create all isomorphisms of a fragment. Thus we do not need an isomorphism test but instead we only need to check for the identity of two graphs, which is much faster. Two graphs are identical if they are isomorphic and if their vertices have the same order.

In mathematical terms a labeled graph $G = (V, E, l_V, L_V, l_E, L_E)$ is *identical* to the labeled Graph $G' = (V', E', l'_V, L'_V, l'_E, L'_E)$ if $\{v_i, v_j\} \in E \Leftrightarrow \{v'_i, v'_j\} \in E'$, $l_V(v_i) = l'_V(v'_i)$ and $l_E(e_k) = l'_E(e'_k)$ hold with $i, j = 1, \ldots, |V| = |V'|$ and $k = 1, \ldots, |E| = |E'|$. The identity is therefore a special kind of isomorphism with $\varphi(v_i) = v'_i$ on page 10 and thus identical graphs are also isomorphic.

The special nature of this isomorphism is that we can use the indices of vertices in order to check the identity of two labeled graphs. If two Graphs G and G' are identical and there is an edge in Graph G from the vertex with the index i to the vertex with the index j, there has to be the same edge $\{v_i, v_j\}$ in G'. So checking for identity is just comparing the edges and their labels, as well as the labels of the vertices, which can be done in a linear amount of time.

Since the core-embeddings are identical graphs it is even enough to check the added part. So for an identity check of, for instance, two edge-vertex pair extensions it is only necessary to compare the labels of the new edges, the labels of the new vertices and the indices of the vertices in the core that are part of the new edges.

The identity check is very fast but it has a flaw. In special cases isomorphic but not identical fragments are created. Consider the graph $x - x - x$. Starting with the core x we will get at the second level a list of four embeddings of the fragment $\tilde{x}-\underline{x}$, where $\tilde{x}$ is the restricted extendable vertex and $\underline{x}$ is the unrestricted extendable vertex. Two of them can be extended at the unrestricted extendable vertex and the others at the restricted extendable vertex, which leads to the extensions $\underline{x} - \tilde{x} - \underline{x}$ and $x - \tilde{x} - \underline{x}$. These extensions are isomorphic but not identical because the index of the vertex in the core that is part of the new edge is different. Thus the sorting algorithm builds two lists of extensions where each of them has two embeddings. The fragments at this level are $\underline{x} - \tilde{x} - \underline{x}$ and $x - \tilde{x} - \underline{x}$. If $x - x - x$ is just a part of a graph the algorithm will continue extending both fragments. It is obvious that the fragments which will be created from the first one are at least those that are created from the second, because the index of the restricted extendable vertex is lower in the first and the extension $\underline{x} - \tilde{x} - \underline{x}$ is thus more extensible than the extension $x - \tilde{x} - \underline{x}$.

The solution to avoid the creation of both extensions is to introduce an isomorphism check for the fragments. If one fragment is isomorphic to another, only that one will be processed with the lower index of the restricted extensible vertex. This fragment is more extensible. For molecular graphs there are just a few not identical extensions of a subgraph, because the number of edges at a vertex is low. This additional isomorphic check will therefore not slow down the algorithm significantly,

because just a few isomorphism checks will need to be done per step. This additional isomorphism check is an improvement of the approach presented in Borgelt and Berthold (2002).

Recursive search Extending the core results in a list of fragments. Each fragment is represented by a list of identical subgraphs. Only frequent fragments are used for further processing. The frequency of a fragment is the number of graphs this fragment is a part of. To determine the frequency of a fragment we iterate through its list of locations (embeddings) and count the number of different graphs. The fragment N − C − C has three embeddings in the database of Figure 2.11. Each embedding is in the same graph and the frequency is therefore equal to one.

All fragments that have a frequency greater than a user specified threshold are used in the continued search. All others can be deleted because the frequency of their supergraphs is even smaller.

A recursive depth first search is started with the first frequent fragment. This means that the extensions of the fragment are created, sorted using a identity check and their frequency is determined with the same method. The algorithm terminates when the entire solution space is traversed.

2.4 Multiple core embeddings

In this section we want to have a closer look at the special case of multiple core embeddings. Multiple core embeddings occur when the chosen core fits more than once into a graph of the database. Consider the graph in Figure 2.13, where B and C are arbitrary substructures.

$$\text{core} - B - \text{core} - C$$

Figure 2.13: An example with multiple core embeddings.

The fragment generation algorithm does work for both embeddings of the core. For the left core the graph looks like a chain and for the right core there is an intersection in the beginning. These are cases that we have discussed in the previous sections and the resulting tree of subgraphs is displayed in Figure 2.14. The subgraphs in the left branch are created from the core at the left and those in the right branches are created from the core at the right.

The problem is that the fragment "core$-B-$core" and its supergraphs are created from both cores. It is just one fragment in Figure 2.14, but if C is a substructure of reasonable size, a lot of fragments will be regarded twice by the algorithm.

A solution for avoiding this overhead would be to use only one core, but there are usually fragments that can be created from one core but not from the other. The subgraph "core $- B$" is such an example that can not be created from the core

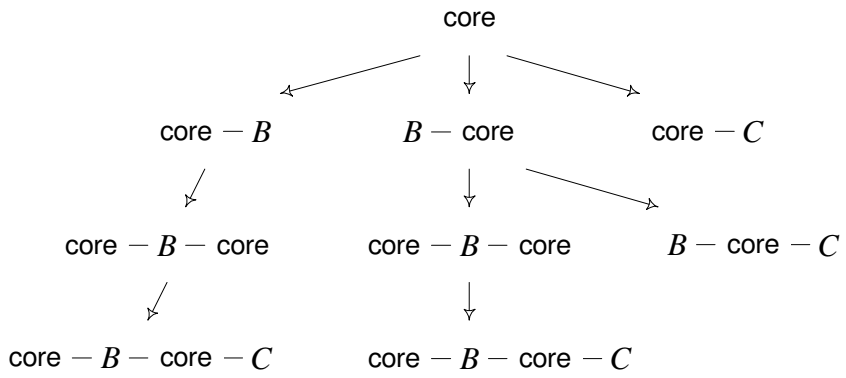


Figure 2.14: The tree of subgraph for the multiple cores example.

at the right. The only possible way out is to avoid such multiple core embeddings. Using bigger cores minimizes the probability of multiple core embeddings.

To get a feeling for the impact of this problem we will regard the molecular graph in Figure 2.15. We have already used this molecule to describe the algorithm itself.

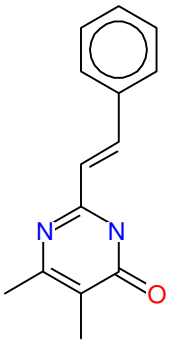


Figure 2.15: A molecular graph.

We perform three runs with the cores O, N and C and we count the number of different fragments at each level in the substructure tree. The results are displayed in Table 2.1. At level one are the cores which are the root nodes and at level 19 are the number of fragments in the substructure tree that are equal to the molecule.

The molecule has 17 atoms but there are 19 levels in the search tree, the reason for the additional levels are the rings. In order to create the whole molecule from one atom we used 17 edge-vertex pair extensions and two edge extensions. At the bottom of the table are the total numbers of fragments created by the algorithm and the number of different fragments.

The search tree for the unique oxygen atom is much smaller than for the others, even though the algorithm is not perfect for the oxygen, because there are equal

Table 2.1: The number of different fragments at one level in the search tree

level	O	N	C
1	1	1	1
2	1	2	6
3	2	8	15
4	4	17	45
5	8	39	95
6	14	77	181
7	24	148	333
8	41	239	545
9	59	315	769
10	67	360	997
11	73	410	1301
12	87	499	1734
13	112	621	2276
14	139	733	2779
15	159	788	3098
16	162	782	3143
17	150	663	2686
18	108	378	1524
19	36	96	384
sum	1247	6176	21912
optimal	727	1607	1663

fragments created. A perfect algorithm would create 727 fragments, which means that MoFa creates in this case circa 1.7 times more subgraphs than needed. Starting with a nitrogen leads to circa 3.8 times more fragments and for the carbon atom circa 13.2 times more fragments are created. This example illustrates clearly the impact of multiple core embeddings, especially for the carbon atom which occurs 14 times in the example.

However the reason for the generation of isomorphic subgraphs in the case of the oxygen atom has to be a different one. The algorithm works fine until the seventh level, it creates all pairwise not isomorphic subgraphs with seven vertices and six edges. Six of those 24 subgraphs are listed in Figure 2.16. They are the subgraphs that will be extended by an edge in the next step.

There is no restriction to edge extensions and extending them generates six isomorphic but not identical subgraphs at the eighth level. Since they are created in distinct branches, none of them will be deleted. A perfect algorithm would generate 36 pairwise not isomorphic subgraphs instead of 41 subgraphs at the eighth level.

The ratio at the ninth level is even worse, there are additional 15 subgraphs. The

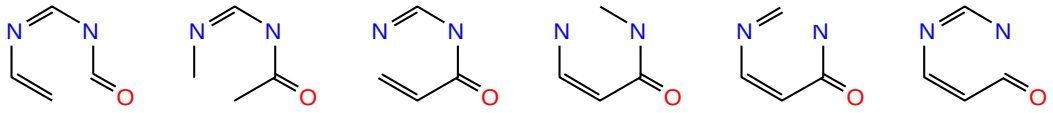


Figure 2.16: Six subgraphs of the molecular graph in Figure 2.11 of the seventh level.

reason for the additional subgraphs at this level are the additional subgraphs at the eighth level. Each of the 6 subgraphs that contain the oxygen and the closed ring are extendable by a carbon atom at three different places, see Figure 2.17. But, since there are 5 additional subgraphs at the eighth level that can be extended to those fragments, there are 15 additional subgraphs at the ninth level.

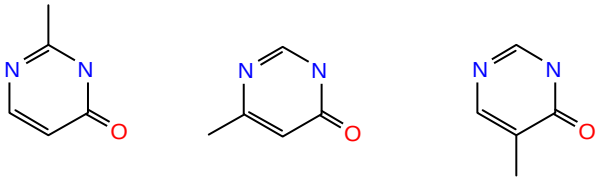


Figure 2.17: Three subgraphs of the molecular graph in Figure 2.11 of the ninth level.

This example has shown again that in case of rings our fragment generation algorithm does produce isomorphic subgraphs. It has also shown that this is a critical point of the algorithm. Because creating some additional fragments at one level leads to an increasing number of additional fragments at higher levels. In the next chapter we will show a solution for this problem

3 Ring extensions

The first topic of this diploma thesis are rings in graphs and their consequences to a fragment mining algorithm. A ring is a closed path in a graph from a graph theoretical point of view. We will restrict the term "ring" to closed paths with five or six vertices. They occur frequent in molecular graphs but the approach can be used for rings with other sizes, too.

The fragment generation algorithm of MoFa has created a ring by adding edge-vertex pairs and one edge in succession to a fragment. The result are several fragments that contain only a part of a ring. This is in contrast to the expectations of chemists and biologists. They are usually interested in fragments that either contain a ring or not. This leads to the idea to combine the extensions that build a ring or in other words only whole rings can be added to fragments.

This has several advantages, the number of fragments will decrease because those fragments that contain only parts of rings will not be created. We have seen that fragments with rings are created several times at different branches of the fragment tree by MoFa. Ring extensions will also reduce this effect and the third advantage is that ring extensions make a special treatment of rings possible. An example are the rings in Figure 3.1. The displayed molecule is benzene, the circle in the middle of the first ring indicates that the ring consists only of aromatic bonds. The second and the third ring is also benzene in a representation with alternating single and double bonds. They are called the Kekule representations of benzene. Ring extensions make it possible that the three representations are treated as equal which is not easy to do in the basic MoFa algorithm.

Figure 3.1: Rings that should be treated as equal.

We assume that the rings are marked in the database. If the rings are not marked it is possible to use the basic MoFa algorithm to find them, by taking a ring as a core and embedding just the core. This core embedding should not make a difference between atoms so that all rings are found even if the core consists only of carbon atoms.

To show the effect of ring extensions we will regard the example in Figure 3.2. The database is the molecule in the box in the upper right corner and we initialize MoFa with the core.

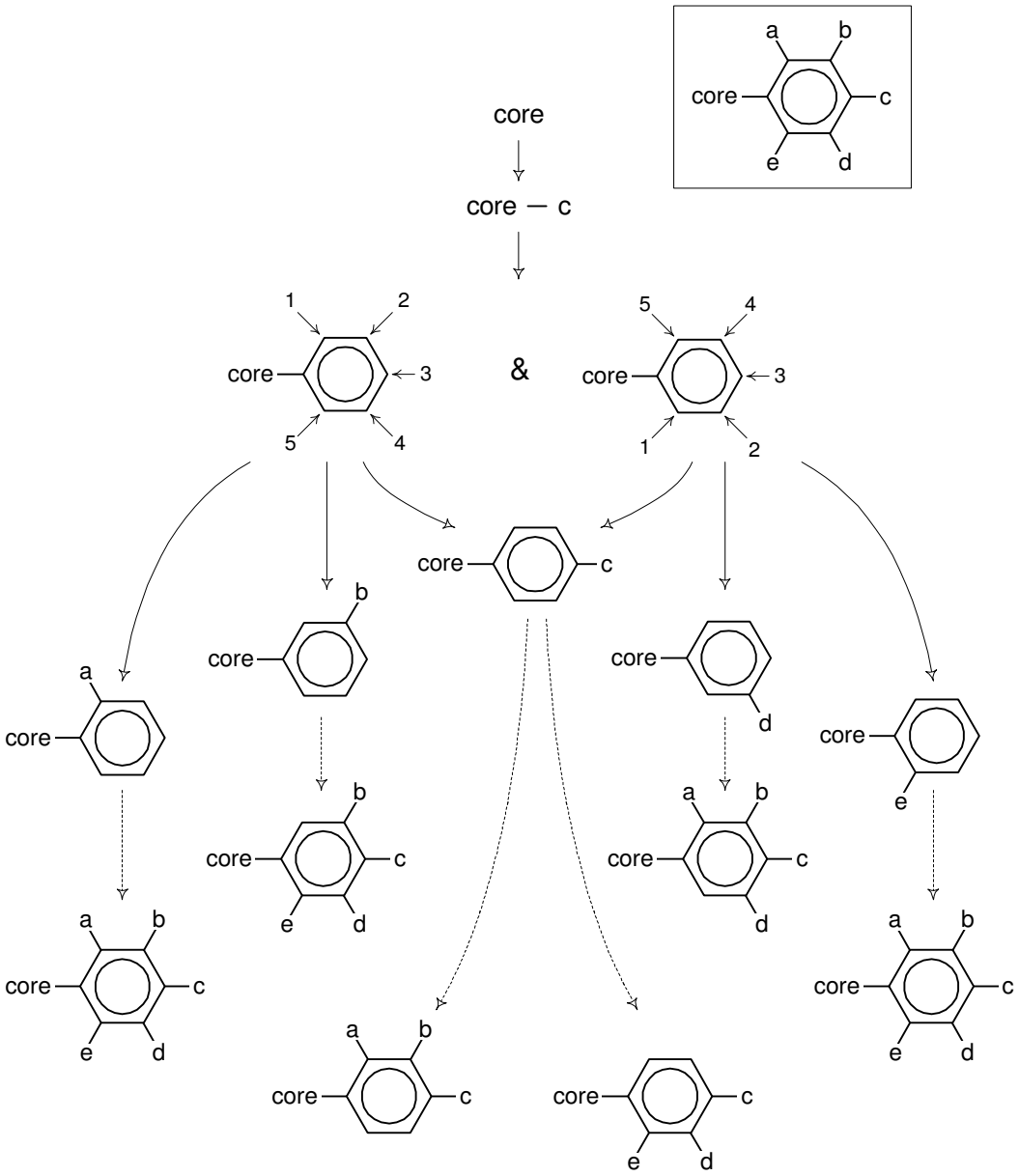


Figure 3.2: The substructure tree for the example in Figure 3.1.

The core can be extended by a carbon atom. We do not add the ring directly to the core. It is easier to implement an algorithm that adds the ring in the second step. Adding the ring means to add the remaining atoms and bonds of the ring to the fragment core $-$ c. Adding the ring is internally done by adding the bonds and the atoms successively. The result are two fragments that are equal in this case of a symmetric ring. They are both displayed in Figure 3.2 where the numbers denote the order in which the atoms have been added. The order is important for

the fragment extension rules in Section 2.2 on page 11.

The carbon that has been added first is marked as the restricted extendable vertex and the remaining atoms of the ring are unrestricted extendable. The next extension is edge-vertex pair '−a'. It can be added to both embeddings of the ring. In the first embedding it is added to the atom with the number one and in the second embedding it is added to the atom with the number five. Those extensions are isomorphic but not identical. The less extendable extension can be deleted. It is the extension of the second embedding. The same holds for the extensions with '−b', '−d' and '−e'.

When we extend the embeddings of the ring with −c we will get embeddings that are extended at the vertex with the number three. They are not distinguishable which is why we get two embeddings of this extensions.

The fragments at the end of the dashed arrows are the largest fragments that are created from one extension of the ring embeddings. The '−c' extension has also two embeddings. There are therefore two fragments created from these embeddings.

There are unfortunately some fragments that are twice in the fragment tree in Figure 2.2. This is an effect of symmetric rings. In the case of asymmetric rings it is possible to distinguish the two embeddings of the ring. They are isomorphic but not identical and the less extendable one is therefore deleted. The consequence is that we get a tree without pairwise equal fragments.

3.1 Frequent fragments in the NCI-HIV database

In this thesis we use a well-known, publicly available dataset from the National Cancer Institute to present results on a real world database. The DTP AIDS Antiviral Screen was developed as an effort to discover new compounds capable of inhibiting the HIV virus. The screen utilized a soluble formazan assay to measure protection of human CEM cells from HIV-1. Full details were published by Weislow, Kiser, Bader, Shoemaker, and Boyd (1989). Compounds able to provide at least 50% protection to CEM cells were retested. Compounds that provided at least 50% protection on retest were listed as moderately active (CM). Compounds that reproducibly provided 100% protection were listed as confirmed active (CA). Compounds not meeting this criteria were listed as confirmed inactive (CI). The dataset consists of 41,316 compounds of which we have used 37,171. For the remaining 4,145 no structural information was available at the time of the experiments reported here. Each compound has a unique number (NSC number). Out of the 37,171 compounds 325 belong to class CA, 896 to class CM and the remaining 35,950 to class CI.

From the 37,171 compounds that we use, there are a small number of compounds (59) listed by the NCI that are known active compounds. They are grouped into seven chemical classes:

- Azido Pyrimidines (10)

Table 3.1: The frequency of the one atomic fragments in *CA*

Atom		frequency in %
Si	Silicon	0.31
Se	Selenium	1.85
I	Iodine	2.46
Br	Bromine	6.46
F	Fluorine	10.46
P	Phosphorus	12.00
Cl	Chlorine	21.23
S	Sulfur	44.00
N	Nitrogen	84.92
O	Oxygen	95.69
C	Carbon	100.00

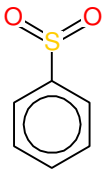
- Natural Products or Antibiotics (8)
- Benzodiazepines, Thiazolobenzimidazoles and related Compounds (10)
- Pyrimidine Nucleosides (11)
- Dyes and Polyanions (15)
- Heavy Metal Compounds (3)
- Purine Nucleosides (2)

We will first run the algorithm to find frequent fragments in the active compounds (*CA*) only. We choose a minimum frequency threshold of 15% and a minimum size of nine atoms to get fragments of reasonable size. The frequency of the single atom fragments is displayed in Table 3.1. Obviously we can only expect to extract fragments containing the five most frequent atoms ($\geq 15\%$).

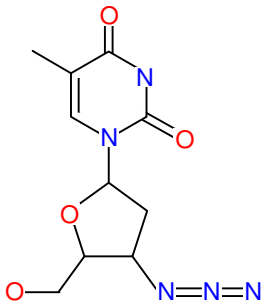
We use the advanced algorithm with ring extensions. It is initialized with the empty subgraph and it takes circa 13 seconds[1] for termination. In that time 456 fragments were created in a tree with a depth of 13 levels. From those 456 fragments 12 are reported that fulfil the constraints. Due to space limitations we have picked out two fragments that are displayed in Figure 3.3.

Fragment 1 is frequent in the group 'Dyes and Polyanions', but it is probably too small to be used as a classification model. Small fragments can usually be expect to be frequent in a all classes and Fragment 1 has indeed a frequency of 16.07% in *CM* and 5.07% in *CI*.

[1] Our experiments were run using a Java implementation of the presented algorithm on a 1GHz Xeon Dual-Processor machine with 1GB of main memory using jre1.4.0.



Fragment 1
freq. in CA: 17.54%



Fragment 2
freq. in CA: 15.08%

Figure 3.3: Frequent fragments in *CA*

Fragment 2 is equal to AZT (#602670) which is a compound in the group "Azido Pyrimidines". AZT is a well-known inhibitor of HIV-1. Therefore one would expect it to be rare in *CI*. The frequency of AZT in *CI* can be determined by initializing MoFa with the core AZT and setting the maximum size to the size of AZT. AZT is, as expected, rare in *CI*, it has a frequency of 0.02% (8 of 35,950) in the inactive compounds.

We have run the algorithm with a relatively low frequency threshold of 15% and the frequencies of the reported fragments are slightly higher. Even without any knowledge about the active compounds, we could infer that the compounds in *CA* are structuraly different. So looking at the structure of the compounds in *CA* we would define several different groups, but the compounds are still in one class because they share the common behavior to provide protection of the CEM cells against the HIV-1 virus. The effect of the structural different groups in *CA* is that we have to use a small frequency threshold.

Search tree We have already mentioned that the algorithm has traversed 456 fragments before termination. 342 of those fragments fulfil the constraints, they are the solution space. The solution space contains 112 pairwise different fragments. We do not report a fragment if it has the same frequency than its children, therefore only 12 fragments are reported. Those 12 fragments are the reduced solution space.

The chart in Figure 3.4 shows the number of nodes in the search tree per level. The tree has a depth of 13 nodes. At the first level is the root node, the maximum width with 77 nodes is at level nine and at the last level are four nodes. The width to depth ration is very high, which is the main reason why we use a depth first search instead of a breadth first search.

When we run the algorithm without using ring extensions we will create much more fragments. It traverses 407,364 fragments before termination which are approximately 900 times more fragments. It needs about 46 minutes for termination

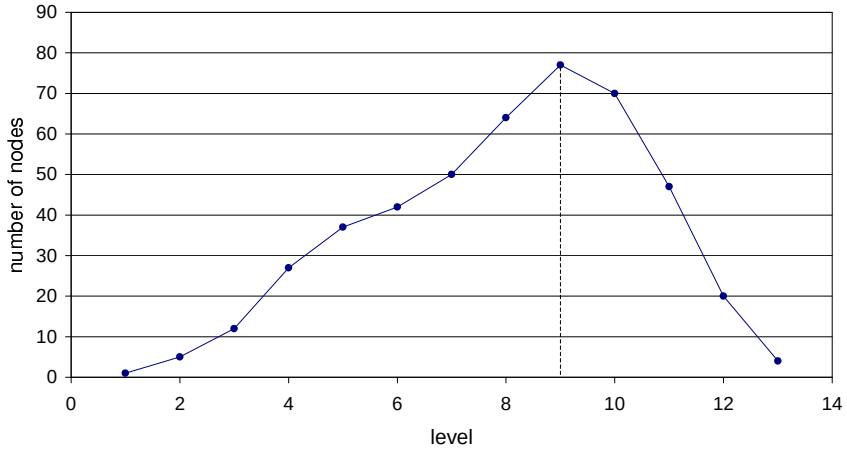


Figure 3.4: The number of nodes per level (using ring extensions).

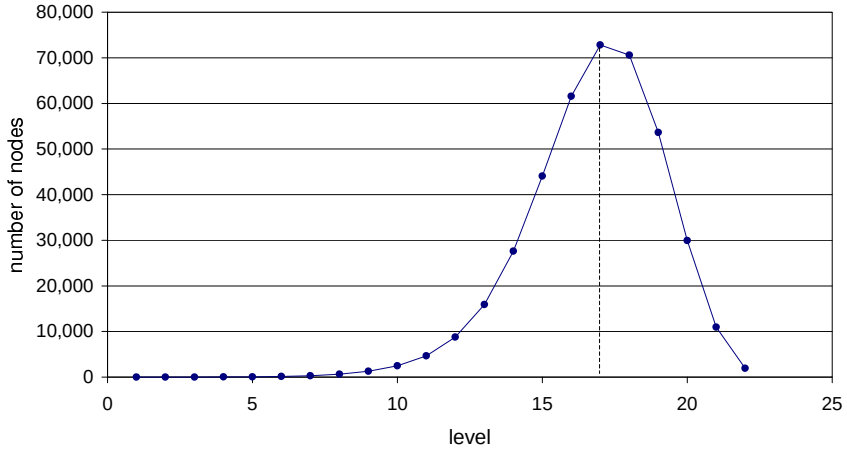


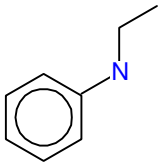
Figure 3.5: The number of nodes per level (without ring extensions).

and the output are 155 fragments. The 143 fragments that are reported in addition are fragments that contain only parts of rings.

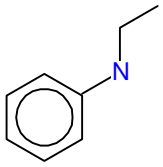
Figure 3.5 is a chart that shows the number of nodes per level for the run without using ring extensions. It starts with one node (the core) at level one. At level eight we already encounter 589 fragments. Those fragments are not part of the solution space, because they are too small. The maximum are 72,825 nodes at level 17 and the last level (22) contains 1944 nodes.

This search tree is deeper because the basic algorithm needs six levels to add a ring with six atoms to a fragment and the advanced algorithm which uses ring extensions needs just one level.

Even if the fragment is found by the basic algorithm and by the advanced algo-

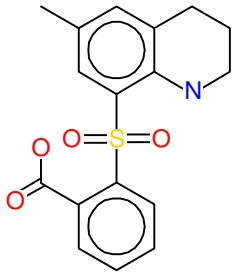


Fragment 3
freq. in CA: 22.77%
basic algorithm

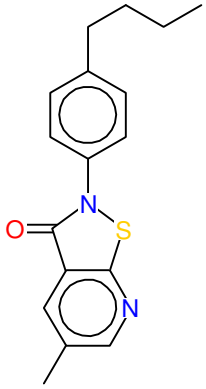


Fragment 4
freq. in CA: 20.00%
with ring extensions

Figure 3.6: Frequencies in *CA* using the basic algorithm and the algorithm with ring extensions.



NSC #667948



NSC #698601

Figure 3.7: Compounds that contain Fragment 3 but not Fragment 4.

rithm it is possible that they report different frequencies. Fragment 3 is an output of the basic algorithm. It has a frequency of 22.77% in the active compounds. Fragment 4 is an output of the algorithm that uses ring extensions. It has a frequency of 20% in *CA*. The fragments are equal so they should have the same frequency in *CA* but Fragment 3 fits into 74 compounds and Fragment 4 fits only in 65 compounds. There are nine compounds that contain only fragment 3, two of them are displayed in Figure 3.7.

In both compounds the $N-C-C$ tail of the fragment is part of another ring. It is not possible for the advanced algorithm to embed the fragment in those compounds because it cannot add a part of a ring. This is a behavior that is expected by chemists and biologists, essentially atoms in rings and chains are considered to be different.

Solution space We have mentioned above that the solution space has 112 fragments whereas MoFa has reported only 12. The reason for this is that MoFa deletes fragments that do not carry new information.

We denote the whole solution space of 112 fragments by $Sol = \{G_{Sol\,1}, G_{Sol\,2}, \ldots, G_{Sol\,n}\}$. A subset of the solution space is the *reduced solution space* defined by

$$R_{Sol} = \{G_{Sol} \in Sol \mid \text{it does no exist a Graph } G \in Sol \text{ with } G_{Sol} \subseteq G \text{ and } \mathit{freq}(G) = \mathit{freq}(G_{Sol})\}\,.$$

We get the reduced solution space by deleting the fragments in the solution space that have a child with the same frequency. It is only necessary to remember the fragments in the reduced solution space, because the frequency of an arbitrary fragment G that is element of the solution space S can be determined by

$$\mathit{freq}(G) = \max\ \mathit{freq}\left(\{G_R \in R_{Sol} \mid G \subseteq G_R\}\right).$$

From the initial 112 fragments of the solution space 12 fragments remain, which are the reduced solution space. Reducing the number of fragments is very important because it makes it much easier for the user to pick out interesting fragments. We will discuss this topic in more detail in the next chapter.

4 Discriminative Fragments and the Version Space

Finding frequent fragments has been discussed in the last chapters. We have taken a set of graphs and we have introduced an algorithm that finds all frequent connected subgraphs. The algorithm can be used to answer various queries. Choosing a frequency threshold of 100%, the result will be the greatest common substructures of the database or for lower frequency thresholds, the fragments that are shared by the majority of the molecules can be found.

On top of this we are interested in fragments that are frequent in one part of the database and rare in a second part. These can be used to distinguish or discriminate between these parts of the database.

A practical example is the NCI-HIV database. The compounds of this database that are in class *CA* inhibit HIV infection and the compounds of class *CI* have no measurable effect.

We first mine fragments that are frequent in the active compounds and rare in the inactive compounds. These fragments are parts of compounds that inhibits HIV infection and are maybe the reason for the activity of those compounds. They might be that part of the compounds that activate the body's defense against the HIV virus. The reason for the activity must obviously still be answered by the biologist. The fragment mining algorithm is only able to suggest fragments that seem to have an impact on the activity of compounds. A well know example of a compound that inhibits HIV infection is AZT (NSC #602670).

From a more theoretical point of view, we consider a database of labeled graphs which is split up into two distinct classes. Fragments that discriminate between the classes can be used as a classification model. New graphs without a class label are assigned to one class if they contain the specific fragments of that class. This methodology is well known as virtual screening for databases of compounds as mentioned in the introduction.

Model The model consists of a database of labeled graphs D and a function $f_C : D \to C$ that assigns each graph to a class of the set $C = \{c_1, c_2\}$. The set $D_i = \{x \in D \mid f_C(x) = c_i\}$ are all graphs that are in class c_i. The algorithm takes a second set of graphs D_F to create the set of candidates. All connected subgraphs of D_F are taken as candidates of fragments that discriminate between D_1 and D_2. The set of connected subgraphs of D_F is the set of fragments donated by F. For a

complete search D_F will be equal to D and for a limited search D_F will be a subset of D or a completely different set of graphs.

Solution space A set of constraints is taken to restrict the search. Monotonic and anti-monotonic constraints are allowed. An example for an anti-monotonic constraint is a lower threshold for the relative frequency $\mathit{freq}(x, D) \geq t\ ,\ t \in [0, 1]$ of a fragment x in the database D.

If a fragment x_2 fulfils this constraint then another fragment x_1 is also frequent when $x_1 \subseteq x_2$. On the other hand when a fragment x_1 does not fulfil the frequency constraint another fragment x_2 with $x_1 \subseteq x_2$ is not rare either. This characteristic is shared by anti-monotonic constraints.

Monotonic and anti-monotonic constraints are defined by their impact on the solution space. The solution space is the set of fragments that fulfil the constraints denoted by $\mathit{sol}(\chi)$ which is a subset of F. An *anti-monotonic constraint* $\overline{\chi}$ is defined by Kramer et al. (2001) as follows:

$$\text{for all } s, g \in F\ :\ g \subseteq s\ \wedge\ s \in \mathit{sol}(\overline{\chi})\ \rightarrow\ g \in \mathit{sol}\,(\overline{\chi})\,.$$

The more general graph g is a subgraph of the more specific graph s which follows the notation of Mitchel (1997). Mitchell considers a space of hypotheses where the general hypothesis allows a general statement. In our terms the supergraphs of s are a subset of the supergraphs of g.

An anti-monotonic constraint $\overline{\chi}$ restricts the solution space at the upper end and a monotonic constraint $\underline{\chi}$ restricts the solution space at the lower end. A *monotonic constraint* $\underline{\chi}$ is defined by

$$\text{for all } s, g \in F\ :\ g \subseteq s\ \wedge\ g \in \mathit{sol}\,\big(\underline{\chi}\big)\ \rightarrow\ s \in \mathit{sol}\,\big(\underline{\chi}\big)\,.$$

Figure 4.1 displays the solution space using anti-monotonic constraints and Figure 4.2 displays the solution space using monotonic constraints in the space of all fragments F. In the lower part are the more general fragments and in the upper part are the more specific fragments.

A conjunction of anti-monotonic constraints is again an anti-monotonic constraint and a conjunction of monotonic constraints is a monotonic constraint, too. This allows to use further anti-monotonic constraints like the maximum size of fragments or the maximum number of aromatic rings in a fragment or further monotonic constraints. Note that the negation of an anti-monotonic constraint is a monotonic constraint and vice versa.

Anti-monotonic and monotonic constraints are important because their solution space is bounded by a border (Kramer et al., 2001). The borders S and G are sets of graphs where the *specific boundary*

$$S = \{g \in F \mid g \in \mathit{sol}(\chi)\ \wedge\ \text{it does not exist a } s \in \mathit{sol}(\chi) \text{ with } g \subseteq s\}$$

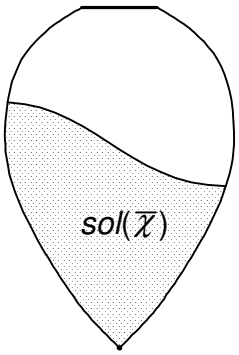


Figure 4.1: lower solution space $sol(\overline{\chi})$, using anti-monotonic constraints

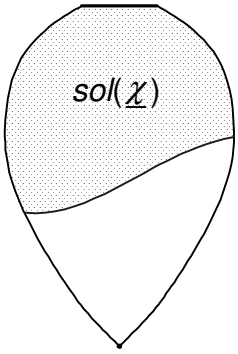


Figure 4.2: upper solution space $sol(\underline{\chi})$, using monotonic constraints

are the most specific graphs that fulfil the monotonic and anti-monotonic constraints $\chi = \underline{\chi} \wedge \overline{\chi}$ and the *general boundary*

$$G = \{s \in F \mid s \in sol(\chi) \ \wedge \ \text{it does not exist a } g \in sol(\chi) \text{ with } g \subseteq s\}$$

are the most general graphs that fulfil the constraints. The solution space with the borders S and G is called *version space* (Mitchel, 1997)

$$\text{VS} = \{x \in F \mid \exists\, g \in G \text{ and } \exists\, s \in S \ : \ g \subseteq x \subseteq s\}\,.$$

The version space VS is equal to the solution space $sol(\chi)$.

Figure 4.3 displays the version space and its borders in the set of fragments. At the lower end is the most general fragment G_0 which is the empty graph. At the upper end are the most specific connected fragments S_0. S_0 is defined by

$$S_0 = \{g \in F \mid \text{it does not exist a } s \in F \text{ with } g \subseteq s\}\,.$$

For the association rule mining task and for an approach that considers also not connected subgraphs, S_0 would be one element. In our case does S_0 usually consist of more than one connected subgraph.

Our algorithm does a bottom up search for finding the fragments in F that fulfil the constraints. The fragments between S_0 and S are created. They are the candidate space F_C but only the fragments beyond G are reported. The candidate space contains the fragments that fulfil the anti-monotonic constraints as shown in Figure 4.4. The monotonic constraints are only used to decide if a fragment should be reported as a part of the version space.

Discriminative fragments Fragments are called discriminative when they are frequent in D_1 and rare in D_2. The algorithm creates the fragments of D_F in a bottom

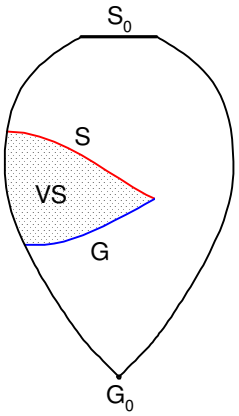


Figure 4.3: version space

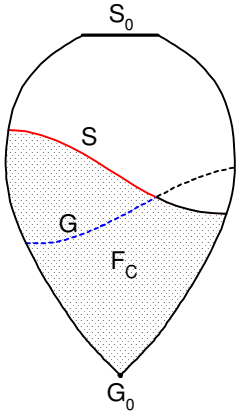


Figure 4.4: candidate space for a bottom up search

up manner and terminates if the criterion $freq(x, D_1) \geq t_1$ is no longer fulfilled. It reports all fragments x with $freq(x, D_2) \leq t_2$.

For computational issues further constraints like the size of the fragments can be used to restrict the solution space.

The reduced solution space As mentioned above it is only necessary to remember the borders S and G to determine the fragments of the version space, because a fragment x is element of the version space if and only if it is a subgraph of a graph in S and a supergraph of a graph in G. But it is not possible to compute the frequencies of x in the databases D_1 and D_2 from the borders S and G.

We have already presented a method in the case of finding frequent fragments in Section 3.1 on page 30. It determines the set of fragments that are needed to compute the frequencies of fragments in the solution space. This set of fragments is a subset of the solution space and is called reduced solution space R_S.

We can extend this approach to the task of mining discriminative fragments. The *reduced solution space* is defined by

$$\begin{aligned} R_{Sol} \;=\; \{g \in \mathrm{VS} \mid &\text{it does no exist a Graph } s \in \mathrm{VS} \text{ with} \\ &g \subseteq s \text{ and } freq(s, D_i) = freq(g, D_i)\ ,\ i = 1, 2\}\,. \end{aligned}$$

The frequency of a fragment in the version space can than be determined by

$$freq(g) = \max\ freq\left(\{s \in R_{Sol} \mid g \subseteq s\}\right)\ ,\ g \in \mathrm{VS}.$$

The reduced solution space contains the upper boundary S of the version space, but not necessarily also the lower boundary G. It is therefore necessary to remember the joint of those sets for creating the version space and determining the frequencies of the fragments in the version space.

With our algorithm the decision if a fragment is in the reduced solution space requires some more computations, because we represent the lattice of subgraphs by a tree. There are some connections in the lattice that are not in the tree. Thus a fragment can have more connections to supergraphs (children) in the lattice than in the tree.

So we have to take care that the supergraphs of a fragment are traversed before the fragment itself. We can then determine the supergraphs of a fragment by substructure checks and add the missing connections.

We achieve this by sorting the extensions of a fragment. They are sorted by their extensibility. An extension is more extensible if the index of its restricted extensible vertex is smaller. In the case of equal indices the attachment of the extension has to be compared. An extension with a lower attachment is more extensible.

Since the extensions are sorted by their extensibility in a descending order from left to right the resulting tree will be deeper at the left (Figure 4.5), it is unbalanced. This unbalanced tree is traversed in a depth first manner starting with the branch at the left.

To illustrate that this sorting and the depth first search really achieve the desired effect, we consider to be at the fragment $N - O$. This node can be extended by a chloride atom. If the rule works, all other fragments that contain $N - O$ are traversed before. We know that at the left of $N - O$ there are fragments with higher extensibility. It follows that the extension $-O$ that lead to $N - O$ could also be added to the fragments at the left. In this case the rule allows $N - C$ to be extended by $-O$ at the nitrogen atom.

For the same reason $N - Cl$ is not extendable by $-O$ at the nitrogen atom. It follows that all supergraphs of $N - O$ are either in the subtree of $N - O$ or in the tree at the left of $N - O$. We can therefore be sure that the substructure test of $N - O$ with the traversed fragments leads to the desired supergraphs of $N - O$.

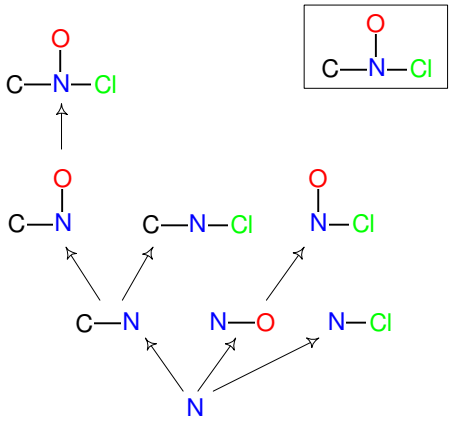


Figure 4.5: Example of a sorted fragment tree.

The substructure test has a further advantage. Our fragment generation algo-

rithm in some cases creates a fragment more than once. If this fragment is in the version space it would be added more than once. Since isomorphic fragments have the same frequencies the second one is not added to the reduced solution space.

4.1 Discriminative fragments in the NCI-HIV database

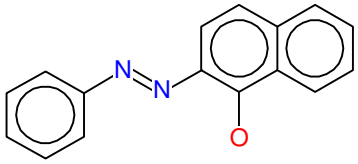
The NCI-HIV database is split up in three classes. Confirmed active *CA*, confirmed moderate active *CM* and confirmed inactive compounds *CI* for the protection of CEM cells against the HIV-1 virus. We want to find fragments that discriminate between *CA* and *CI*. We use the MoFa miner that considers ring extensions. We choose a minimum frequency threshold for *CA* of 15% which is the same as in Section 3.1 and for class *CI* we take a maximum frequency threshold of 0.1%. The result are 15 fragments, which is AZT itself and substructures of AZT. AZT is equal to Fragment 2 on page 27 which is frequent in *CA*. It covers 49 of 325 (15.08%) compounds in *CA* and only 8 of 35,950 (0.02%) compounds in *CI*. AZT belongs to the group of azido pyrimidines which is a well known group of chmicals that inhibits HIV infection.

Fragment 1 on page 27 is with 17.54% more frequent than AZT in the active compounds. But it is not one of the 15 disriminative fragments because it covers 57 of the active but also 1822 (5.07%) of the inactive compounds. Fragment 1 is an example of a frequent fragment in *CA* that can not be used as a classification model because it is also frequent in *CI*.

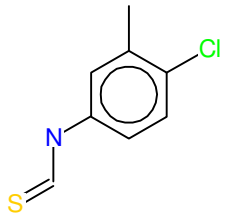
To get more discriminative fragments we have to lower the minimum frequency threshold in *CA* or to raise the maximum frequency threshold in *CI*. It is hard to define which are the optimal thresholds. The difference between the thresholds has to be significant and the number of covered compounds has to be high enough to avoid stochastical effects.

Fragment 5 is not one with the highest difference of the frequencies in *CA* and *CI* but it is also frequent in the group “Dyes and Polyanions”. It is in 12 of 15 compounds of this group. This is the second fragment that belonged to one of the groups on page 25. The groups ’Heavy Metal Compounds’ and ’Purine Nucleosides’ are to small and the remaining ones are structurally inconsistent.

Fragment 6 is remarkable because it is not covered by any compound of the 35,950 compounds in *CI* and the fact that it covers 31 compounds in *CA* makes it a very good classification model. However, it is not covered by any of the known active groups.



Fragment 5
freq. in CA 5.23%
freq. in CI 0.08%



Fragment 6
freq. in CA 9.85%
freq. in CI 0.00%

Figure 4.6: Discriminative fragments.

5 Fuzzy Fragments

Until now we have regarded a method for finding fragments in a molecular database that fulfil given constraints. An important constraint is the minimum frequency of a fragment. This constraint builds an upper border of frequent fragments where fragments beyond this border are not frequent enough (see Chapter 4). We traverse the lattice of fragments in a depth first manner and every time we reach this border we stop following this branch and we proceed with another branch.

We will consider next a new aspect for the search of fragments. Chemists and biologists regard some molecules (fragments) as equal although they are structurally different. If there are for instance two fragments that differ just in one atom. They could, under certain circumstances, be considered as one meta or fuzzy fragment.

So a fuzzy fragment is a union of fragments. If we want to find fuzzy fragments we first need rules that define which fragments should be grouped together and secondly we have to check during the run of the algorithm if the fragments match using the fuzzy rules.

We allow a fuzzy match if the fragments differ in a maximum number of atoms. Atoms that match together have to be part of the same atom group where we distinguish between atom groups for atoms in rings and atom groups for atoms outside of rings.

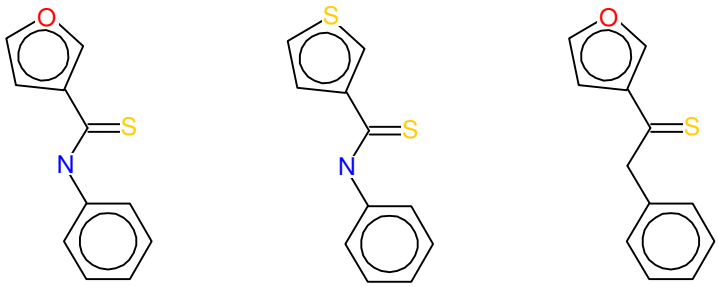


Figure 5.1: Example of three fragments that can be grouped together to form fuzzy fragments.

Let us consider the example in Figure 5.1. There are three fragments of a fictitious database. When we consider two atom groups, the first is represented by A_r and contains oxygen and sulfur but they match only if the oxygen atom and the sulfur atom is part of a ring. The second atom group A defines atoms that match if

they are not part of a ring. It contains nitrogen and carbon. We use the following notation:

$$\begin{array}{lcl} A_r : O, S & \rightarrow & O \text{ and } S \text{ match if they are part of rings.} \\ A \ : N, C & \rightarrow & N \text{ and } C \text{ match if they are not part of rings.} \end{array}$$

When we observe the fragments in the example for fuzzy matches with one fuzzy atom, we see that the fragment at the left and the fragment in the middle fit together. They differ only in the atom at the top. It is part of a ring and the oxygen and the sulfur are in a "ring atom group".

The fragment at the left also forms a fuzzy union with the fragment at the right. The difference is the nitrogen and the carbon in the chain between the rings and those atoms are in atom group A.

We have allowed only one fuzzy match of atoms. This is the reason why the fragment in the middle and the fragment at the right do not match. They differ in two atoms.

We have chosen a fuzzy match of fragments that concentrates on atoms. It follows that only fragments with the same size and the same structure can form a fuzzy union. When we build the lattice of fragments in such a way that one level contains fragments with the same size, we can restrict the task of finding fuzzy unions in the lattice to the task of finding fuzzy unions in one level of the lattice. However, we still have to expect that one fragment is part of several fuzzy unions like in the example.

We have explained that a fuzzy match or fuzzy fragments are a union of fragments. There are maybe fuzzy unions of fragments that are beyond the border of frequent fragments. But the combination of rare fragments that do not satisfy the minimum frequency criteria can still lead to a fuzzy fragment that fulfils this criteria. So finding fuzzy fragments must include also fragments that are beyond the usual border of frequent fragments and it does therefore raise the size of the problem.

5.1 The fuzzy approach

The fuzzy approach is a method to find unions of fragments that contain similar fragments. We will switch back to graph theoretical terms in this section to show that the approach is not restricted to molecular graphs.

We restrict the fuzzy match to the match of vertices (atoms). Vertices match if their labels are equal, they do fuzzy match if their labels are equal or if their labels are in a "vertex label group". When we consider two labeled graphs $G = (V, E, l_V, L_V, l_E, L_E)$ and $G' = (V', E', l'_V, L'_V, l'_E, L'_E)$ we have the set of labels $L_V \cup L'_V$. A *vertex label group* is a subset of this set. We will usually have a set of vertex label groups $V_g = \{V_{g1}, V_{g2}, \ldots, V_{gn}\}$ where $V_{gi} \in L_V \cup L'_V$.

Using this definition we can say that *vertex v is fuzzy equal to vertex v'* if

$$l(v) = l'(v') \text{ or if it exists a } V_{gi} \in V_g \text{ so that } l(v) \in V_{gi} \text{ and } l'(v') \in V_{gi}.$$

We will denote the fuzzy equality of the vertices v and v' by $l(v) \cong l'(v')$.

Subgraphs that are fuzzy equal have a maximum number of different but fuzzy equal vertices. Such subgraphs can be grouped together to a *fuzzy union*. A *fuzzy union* of graphs is in mathematical terms a set of pairwise *fuzzy isomorphic* graphs where two labeled graphs G and G' are *fuzzy isomorphic* if there exist a isomorphism φ so that G and $\varphi(G')$ are *fuzzy identical*.

Fuzzy identical graphs have the same size and structure and differ only in the labels of some vertices. Two graphs G and G' are called *fuzzy identical* or *fuzzy identical of degree d* if and only if the conditions

1. $\{v_i, v_j\} \in E \Leftrightarrow \{v'_i, v'_j\} \in E'$
2. $l_E(e_k) = l'_E(e'_k)$
3. $|\{i \,|\, l(v_i) \ncong l'(v'_i)\}| = 0$
4. $|\{i \,|\, l(v_i) \neq l'(v'_i) \text{ and } l(v_i) \cong l'(v'_i)\}| \leq d$

hold with $i, j = 1, \ldots, |V| = |V'|$ and $k = 1, \ldots, |E| = |E'|$. The first condition is fulfilled if the graphs have the same structure and the second demands the equality of the edge labels. The third condition means that there are no vertices that are neither equal nor fuzzy equal. And the last condition limits the number of vertices that are not equal but fuzzy equal.

We have now described mathematically what we understand by fuzzy fragments. We have explained in the introduction of this chapter that a fuzzy fragment is a union of fragments at one level in the lattice of subgraphs. The task is therefore to find the fuzzy isomorphisms at a given level.

The molecular fragment miner creates the subgraphs of all spanning trees in the database. This was an important fact for graphs with rings in Section 2.2 on page 12. In other words the tree of subgraphs contains all isomorphisms of a fragment. Since we create all isomorphisms we can use an identity check of fragments instead of an isomorphism check to find fuzzy unions.

Using the fragment generation of MoFa we could collect the fragments at one level and we could determine fuzzy fragments by conducting pairwise fuzzy identity tests. The problem is that we perform a depth first search. So collecting fragments for each level means to remember the whole search tree, because we do not know all fragments at one level until the algorithm is in the last branch. We have shown examples in earlier chapter that the search tree can be very big, it is therefore not feasible to hold all fragments in main memory at the same time.

Using MoFa we have only those fragments of one level in main memory that are created from one node of the lower level. So we have to make sure that we can still build all fuzzy unions at one level by comparing those fragments. In other words if two fragments at one level do fuzzy match they have to be created from one node at the next lower level so that we can recognize this fuzzy match. Since we create all isomorphisms of a fragment it is enough to compare the children of a node. But

we have to consider that MoFa does not create all children of a fragment. It only creates the fragments that fulfil the rules in Section 2.2 on page 11.

Thus we have to check if those rules remove children that are potentially part of a fuzzy union. The first rule is that fragments can not be extended at atoms with an index lower than the index of the restricted extendable vertex. The fragments that are removed by this rule have been extended at an atom with a different index than the remaining ones. They are certainly not fuzzy identical to the remaining fragments because the first condition (structural identity) for the fuzzy identity is not fulfilled.

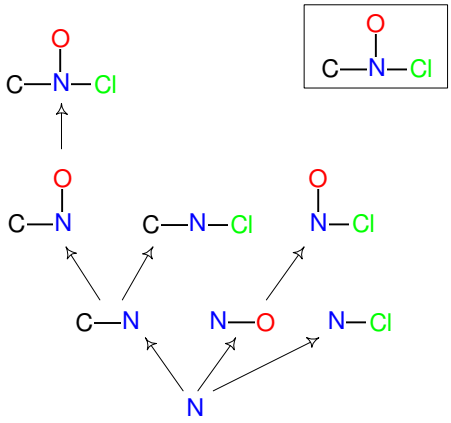


Figure 5.2: Example of a fragment tree.

The second rule restricts the possibility of extensions at the restricted extendable vertex. In the example in Figure 5.2 it avoids that $N-O$ is extended to $C-N-O$ because this fragment is already created in the branch at the left. But when we consider the atom group $A : C, Cl$ we would not recognize the fuzzy union $A-N-O$ because $C-N-O$ and $Cl-N-O$ are created in different branches of the fragment tree.

The solution to this problem is the modification of the second rule. We have to create the fragment $C-N-O$ twice but the second time is just to build fuzzy unions. We therefore call the extension of $N-O$ to $C-N-O$ a *weak extension*. An edge-vertex extension is weak when it extends the core at the restricted extendable vertex, when the attachment criteria is not fulfilled and when the label of the new vertex is in a vertex label group.

A weak extension can only survive in a fuzzy union where the fuzzy union contains at least one non-weak extension. This avoids in the example that the subtree of $C-N-O$ is created twice. When we use this modified rule we get the tree of fragments as displayed in Figure 5.3.

At last we have to define the attachment of the fuzzy extension '$-A$'. Its attachment is equal to the maximum attachment in its vertex label group. It is hence not possible to extend $N-A$ to $O-N-A$ using this rule.

The example does not contain a fragment with more than one fuzzy atom. Such fragments occur when a fuzzy fragment is extended or a ring extension can have more than one fuzzy atom.

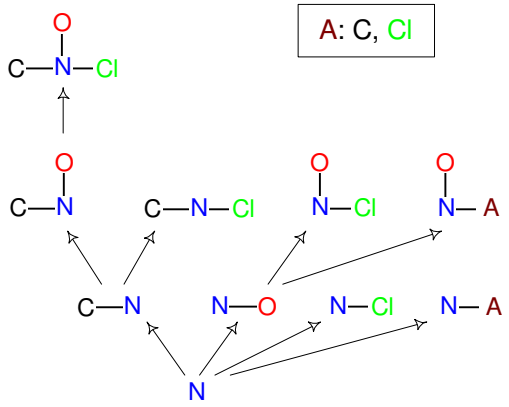


Figure 5.3: Example with fuzzy fragments.

5.2 Fuzzy fragments in the NCI-HIV database

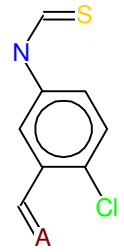
We have used the NCI-HIV database to give examples for frequent fragments in Section 3.1 and discriminative fragments between the active (CA) and the inactive compounds (CI) in Section 4.1. In this section we want to show that fuzzy fragments that satisfy our definition are of practical relevance.

Let us consider the first fuzzy fragment in Figure 5.4. The fragment has one fuzzy atom that can be replaced by oxygen or nitrogen. The fragment with an oxygen at that position has a frequency of 5.5% in the active compounds and the fragment with a nitrogen at that position has a frequency of 3.7% in the active compounds. This gives a total frequency of 9.2% of the fuzzy fragment. It has a frequency of 0% in the inactive compounds which makes this fuzzy fragment to be a discriminative fragment of high importance.

Only the fuzzy approach makes it possible to find this fragment, because when we run the algorithm with a minimum frequency threshold of 5% we will find the first but not the second fragment in this fuzzy union. A run with a frequency threshold of 3% would give us both but also a lot of other fragments and we would not be sure that there is no other fragment with a frequency lower than 3% that could fit in a fuzzy union with the first and the second.

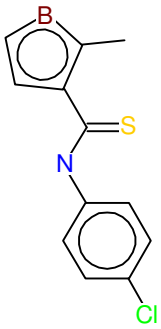
The fuzzy approach allows us to run the algorithm with a frequency threshold of 9% to find Fuzzy Fragment 1. The high frequency threshold avoids the output of a lot of uninteresting fragments and we are sure every fragment is found that fits in the fuzzy union. Fuzzy Fragment 2 also consists of two fragments. It is an example

for a fuzzy fragment with a fuzzy atom which is part of a ring and it shows that also members in a fuzzy union with an extremely low frequency are found.



Fuzzy Fragment 1

A :	O	N
freq. in CA	5.5%	3.7%
freq. in CI	0.0%	0.0%



Fuzzy Fragment 2

B :	O	S
freq. in CA	5.5%	0.01%
freq. in CI	0.0%	0.0%

Figure 5.4: Fuzzy fragments in the NCI-HIV database.

6 Design and Implementation

In the last chapters an algorithm has been discussed that is able to find discriminative fragments in a database of graphs. On top of this a fuzzy comparison of graphs was introduced to fulfil the needs of real world data. In this chapter we describe the implementational details of the presented approaches. We have chosen an object oriented implementation in Java version 1.4. A good introduction to the java language is given by Eckel (2000).

One key feature of the Java language are *interfaces*. They define the functions that an object has to provide. The interface `Graph` contains five functions. The first returns a unique identification, the second and the third are access functions to the elements of a graph and the last functions return the size of the graph itself.

```
public interface Graph {
  /** returns a unique identification   */
  public String getID();

  /** returns the node at index   */
  public GraphNode getNode(int index);

  /** returns the edge at index   */
  public GraphEdge getEdge(int index);

  /** returns the number of nodes  */
  public int getNodeCount();

  /** returns the number of edges  */
  public int getEdgeCount();
}
```

It is possible to write algorithms in Java which use interfaces like `Graph` instead of fixed classes. The major advantage is that the implementation of `Graph` can be changed without any changes in the algorithm that uses this interface. It is for instance possible to use a memory optimized implementation for a database with huge graphs and for databases with small graphs a speed optimized implementation might be more suitable.

`GraphNode` and `GraphEdge` are also interfaces they encapsulate the basic elements of a graph. `GraphNode` provides functions to access its neighbors and `GraphEdge`

consists of two nodes that are connected by this edge. The interfaces and classes that are related to graphs are in the package tripos.sf.MoFa.Graph, where the package MoFa contains the classes that are needed for running the molecular fragment mining algorithm itself.

6.1 Package Graph

The package *Graph* contains classes that work with basic graphs. The class diagram of the basic interfaces is displayed in Figure 6.1. The upper part of the figure illus-

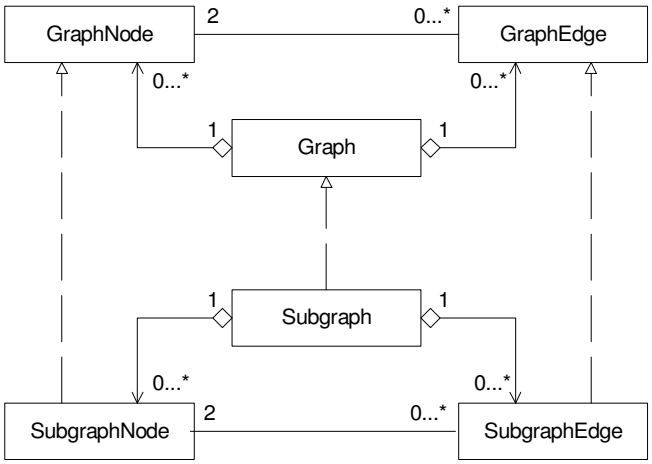


Figure 6.1: The class diagram of the graph related interfaces.

trates, that a `Graph` consists of a `GraphNode` collection and a `GraphEdge` collection, whereas a `GraphNode` has references to a collection of `GraphEdge`s and a `GraphEdge` has references to two `GraphNode` objects. This is identical to the definition of a graph on page 9, where a graph consists of a set of vertices and a set of edges and an edge is a set of two vertices.

A similar structure exists for the `Subgraph` interface. The dashed arrow between `Graph` and `Subgraph` means that a `Subgraph` is a `Graph`, it adds additional functions to the `Graph` interface. In this case the `Subgraph` adds a function that returns a reference to its underlying `Graph`. The same holds for `SubgraphNode` and `SubgraphEdge`, they have references to there counterparts in the underlying graph. Theses references define the location of the subgraph in the parent graph.

The fact that a `Subgraph` is a `Graph` is very useful, because algorithms that work with the `Graph` interface can also be used with classes that implement the `Subgraph` interface.

GraphEmbedder

The `GraphEmbedder` determines all locations of a core in a graph. The core and the graph are both `Graph`s and the result is a list of `Subgraph`s, which are identical

subgraphs that are isomorphic to the core.

The idea of the `GraphEmbedder` is to create a subgraph of the core starting from a single atom and to keep track of the graph during the growth process. In Chapter 4 on page 31 it is explained that we distinguish between the set of labeled graphs that are used to create fragment candidates and the database itself. In this case the core is used to create fragments and the graph is the database.

The algorithm starts by creating a subgraph of the core that contains the first vertex of the core. This subgraph is embedded into the graph. In this case it is just an easy search over all vertices and a subgraph is created containing a vertex of the graph if this vertex is equal to the first vertex of the core. We have now one embedding in the core and a list of embeddings in the graph of the same vertex.

Next a recursive search is started. First, the core embedding is extended using a `SubgraphExtender`, one of those extension is selected, which becomes the new core embedding. Second, the core embedding is embedded into the graph using a `TemplateSubgraphExtender`. These steps are done until the core embedding contains all vertices and edges of the core. In the end there is a core embedding that is isomorphic to the core and there are graph embeddings that are identical to the core embedding and isomorphic to the core.

It is only necessary to regard one subgraph of the core, because this core embedding is isomorphic to all other possible core embeddings and therefore the subgraphs that are identical to the core embedding, are isomorphic to the other possible core embeddings.

SubgraphExtender

The `SubgraphExtender` is for instance used by the `GraphEmbedder` to extend the core embedding. It takes a `Subgraph` which is called seed and returns the possible extensions of the seed. Extensions are supergraphs of the seed and subgraphs of the seed's parent. There are four extensions implemented in the current version. An extension is, like the seed, a connected labeled graph.

- *node extension*: Occurs only when the seed is the empty graph. It adds a node[1] of the seed's parent to the seed.
- *edge extension*: Adds an edge to the seed. The nodes that are connected by this edge are part of the seed. Since the seed is a connected graph, the additional edge closes a ring.
- *edge-node extension*: Adds an edge-node pair to the seed. This extension lets the seed grow by one node.
- *ring extension*: The ring extension is the only used substructure extension. In molecular graphs rings with five resp. six nodes str considered as one

[1] In the implementation we have chosen the term "node" instead of "vertex".

part. Nodes that are in rings have special markers which are used by the `SubgraphExtender` to create the ring extensions.

The nodes and edges of the seed's parent are marked with -1 if they are not in the seed and marked with their index in the seed otherwise. We do check for each node in the seed if its parent is connected to edges that has a marker to -1. These edges are used in extensions. They are the possible extension directions. After that, different extensions are created depending on the nodes of extension directions. If both nodes are in the seed an edge extension will be created. If one node is not in the seed and both nodes are not in the same ring an edge-node extension will be created and finally if one node is not in the seed and both nodes are in the same ring a ring extension will be created.

TemplateSubgraphExtender

The `TemplateSubgraphExtender` is for instance used by the `GraphEmbedder` to keep track of the extension of the core embedding in the graph. It takes an extension of a seed which is called template extension and a subgraph. The subgraph has to be identical to the seed. It tries to create an extension of a subgraph that is equal to the template extension.

Suppose that the template extension is an edge extension. The edge extension is defined by the indices of two nodes and the new edge. To create the same extension there has to be an edge between the same nodes in the subgraph that is in the subgraph's parent but not in the subgraph. The same marking method is used as in the `SubgraphExtender` to determine the nodes and edges that are in the subgraph. If there is an edge between the nodes an edge extension of the subgraph is created. Similar functions exist for edge-node extensions and ring extensions.

The `TemplateSubgraphExtender` does a directed search for extensions. It is used to speed up the algorithm.

Factories and Comparators

The algorithms in the package `Graph` run with interfaces instead of particular classes, as mentioned in the introduction of this chapter. It follows from this that the `GraphEmbedder` cannot create a subgraph since there is no default implementation of a subgraph that is known by the `GraphEmbedder`. We have chosen the design pattern *factory*[2] to create the needed objects. It is for instance required to pass an `EdgeExtensionFactory` to the constructor of `SubgraphExtender`. This factory is an interface that contains the function `createEdgeExtension` which returns an `EdgeExtension`. This factory function takes as arguments the seed, the new edge and the indices of the nodes that are part of the new edge. The factory can return any object as long as it contains the functions that are defined in the `EdgeExtension` interface.

[2]More information about design patterns can be found in (Eckel, 2002).

Another implementational detail is that comparators are used to compare nodes and edges. This allows the client programmer to change the comparison behavior in an easy way during runtime. It is for instance possible to make not a distinction between nodes if they belong to the same group. In case of molecular data it makes for instance sense to try a run without making a distinction between the atoms that belong to the group of halogens.

Using factories and comparators is the final step to decouple the algorithms in the package `Graph` from a particular implementation.

6.2 Package MoFa

The package MoFa finally contains the implementations of the interfaces that are defined in the package Graph. `BaseAtom`, `BaseBond` and `BaseMolecule` are implementations of the interfaces `GraphNode`, `GraphEdge` and `Graph`.

In addition there are the mining algorithm `MoFaMiner`, the graphical user interface `MoFaMinerGUI` and utility classes like `MoleculeParser` and `RingMiner`. The `MoleculeParser` creates a `BaseMolecule` from a string representation of a molecule and the `RingMiner` marks all rings of size five and six in a list of molecules.

The most interesting class is the `MoFaMiner` which uses the algorithm in the package `Graph` for mining discriminative fragments. It takes two sets of molecules called *focus* and *complement*. A minimum frequency threshold is assigned to the focus and a maximum frequency threshold is assigned to the complement. Hence we will mine fragments that are frequent in the focus and rare in the complement.

The focus is used for creating the fragments, because fragments that can not be embedded in one molecule of the focus have a frequency of zero in the focus and are therefore not interesting. Using the focus for creating the fragments makes sure that only fragments are created that have a chance to fulfil a minimum frequency threshold that is greater than zero.

Using an empty complement it is also possible to mine only frequent fragments in the focus.

Tree traversing

After reading the data from a file, the `MoFaMiner` takes the user specified core and embeds it into the focus and into the complement. The core is the root of the search tree, whereas this search tree can be traversed in different ways. We use a pure depth first search. A pure breadth first search is not advisable, because the width to depth ratio of the search tree is high. A combination of both could be used in future implementations instead of the pure depth first search. The decision about the traversing behavior is done by a *strategy* design pattern called `TraversingStrategy`.

The `TraversingStrategy` decides in which order the nodes of the search tree should be traversed, it does not store any data. The `TraversingStrategy` is also

an interface, because we want to use different implementation in future. The implementations of the interface `TraversingModel` is used for holding data.

After the call of the `startTraversing()` function of the strategy it asks the model for the root node. It then uses a `NodeFactory` to create the children of the root. It next goes to one of the children to extend this child and it informs the `TraversingModel` about each move. The `TraversingStrategy` tells the `TraversingModel` which node it leaves and which node it enters and uses the `NodeFactory` to create new nodes.

The `TraversingModel` can create the whole search tree from the information that is given by the `TraversingStrategy`. It can therefore depend its decision if a node (fragment) should be stored not only on the node itself but also on its parent and children.

The beauty of this design is that different traversing strategies can be used with different implementations of the `TraversingModel` and that a model can be used with different implementations of the `TraversingStrategy`.

This does not hold for the `NodeFactory` and the `TraversingModel`. The model has to know which nodes the factory creates. A node is in this case represented by the empty interface called `NodeMessenger`. The `NodeMessenger` is a *messenger* design pattern. In the interface `NodeMessenger` no functions are defined, because it should not be restricted. It is meant as an identifier of a node in the search tree. This could be an object that holds all information about the node but it could also be a key for a map of all nodes of the tree.

The interfaces described above and the class `DepthFirstStrategy` are in the package tripos.sf.MoFa.Tree. In the MoFa package is a `MoFaNodeMessenger` that implements `NodeMessenger` and a `MoFaTraversingModel` that implements the interface `TraversingModel` and the interface `NodeFactory`. We have chosen to put the factory and the model in one class resulting in a more efficient implementation.

6.3 The user interface

We also implemented a graphical user interface for the MoFa miner that allows to read data, set parameters, run the miner and explore the results.

Figure 6.2 shows the main window of the `MoFaMinerGUI` and the dialog where focus and complement can be defined.

The user can specify the seed (core) by typing a string representation of the fragment in the upper right text field. The formats SMILES and SLN can be used. SMILES is a standard from daylight[3] and SLN is a standard from Tripos Inc. The current seed is shown in the area in the middle.

The next four text fields allow to set monotonic and anti-monotonic constraints. There are the minimum frequency in the class focus, the maximum frequency in the class complement and the minimum resp. maximum size.

[3] see www.daylight.com

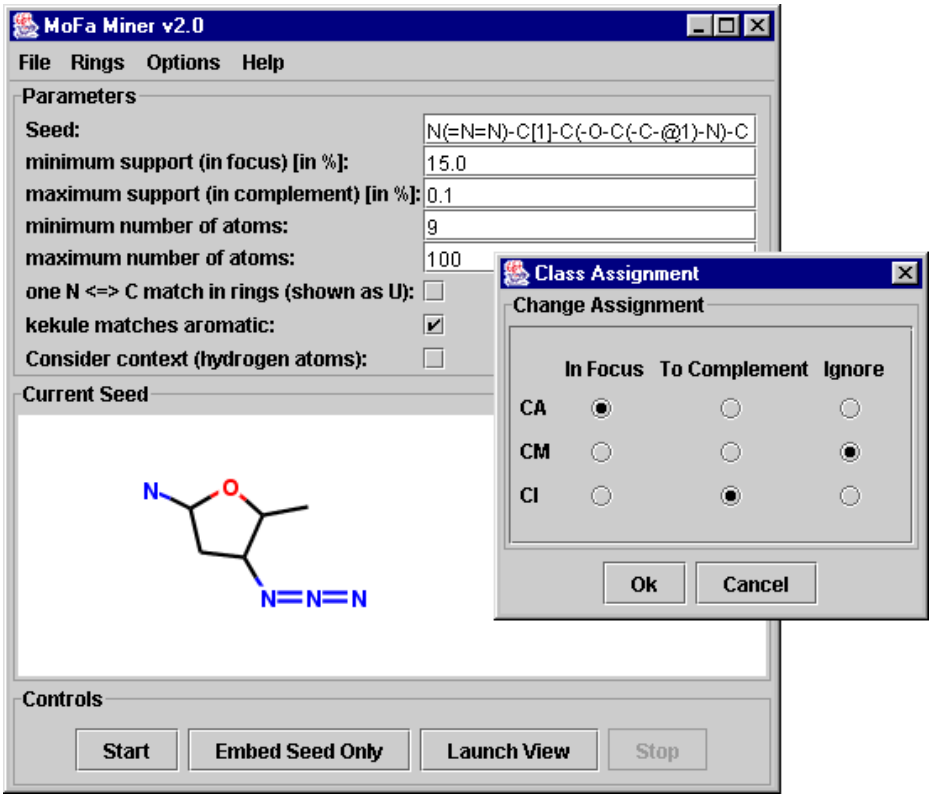


Figure 6.2: The main window of the MoFa GUI.

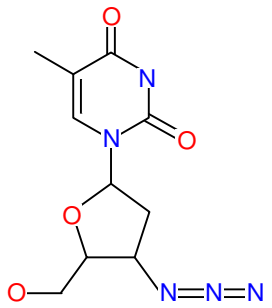
The last three check boxes control special behaviors of the algorithm. Selecting the first allows fragments with one fuzzy match of a nitrogen atom and a carbon atom in a ring. An uranium atom is used to mark this match in the results.

The second box is checked by default. If checked, kekule and aromatic rings with six atoms match.

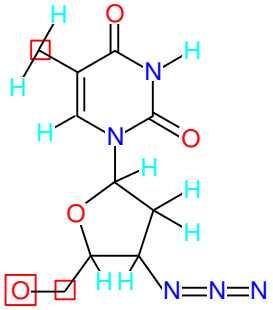
Checking the third box, hydrogen atoms are considered during the search. Hydrogens are usually not displayed, because the number of hydrogen atoms connected to an atom can be calculated from the difference of its valency number and the number of atom pairs to its neighbors.

Figure 6.3 shows the output of MoFa without considering hydrogen atoms at the left and at the right is the counterpart fragment where hydrogen atoms are used. The framed atoms at the right have free valences. That indicates that these are atoms from which the fragment could grow in its underlying molecules. Those atoms are the sections where the fragment is cut out of the molecule. The drawback of considering hydrogen atoms is that the search tree gets bigger which slows down the search significantly.

The dialog with the name ”Class Assignment” allows the user to specify which



ignoring hydrogens
freq. in CA 15.08%
freq. in CI 0.05%



using hydrogens
freq. in CA 15.08%
freq. in CI 0.05%

Figure 6.3: Example for the impact of the "Consider context" flag.

classes are in the focus, in the complement or are ignored. The situation is shown for the NCI-HIV data. The active compounds are in focus, the moderate active compounds are to be ignored and the inactive compounds are in the complement.

The start button at the lower left corner starts the search. Some information is provided to display the progress and a dialog informs about the termination. After termination the "Launch View" button opens the window in Figure 6.4. It shows the fragments that contain the core and that fulfil the constraints. The fragments are numbered starting from one. Information about the chemical structure, size and frequencies in focus and complement are provided next to the list of fragments.

A better view of the structural dependencies is provided by the fragment tree in Figure 6.5. The left part shows the fragments number and the frequencies in focus and complement. The right part shows the chemical structure of the fragments.

The fragment list is a view that can be used to display different information about a fragment, where the fragment tree displays structural dependencies and can be used to group the fragments and to find structurally interesting fragments.

Fragment 3 is selected in the fragment tree. Selecting a fragment causes those compounds which contain the fragment to be shown in the "Structure View" (Figure 6.6). The first compound has the NSC number 633772 and is part of *CA* and the second has the NSC number 640323 and is in *CI*. The classes are indicated by the colored bars at the right of the NSC numbers.

There is one button of the main window that has not been mentioned yet. The "Embed Seed Only" button embeds the seed (core) only, without any attempt to grow the fragment. It uses the MoFa miner but it changes automatically the minimum support to 0% the maximum support to 100% and the minimum size resp. maximum size to the size of the core. 'Embed Seed Only' is a fast way to get statistics for a fragment in a database.

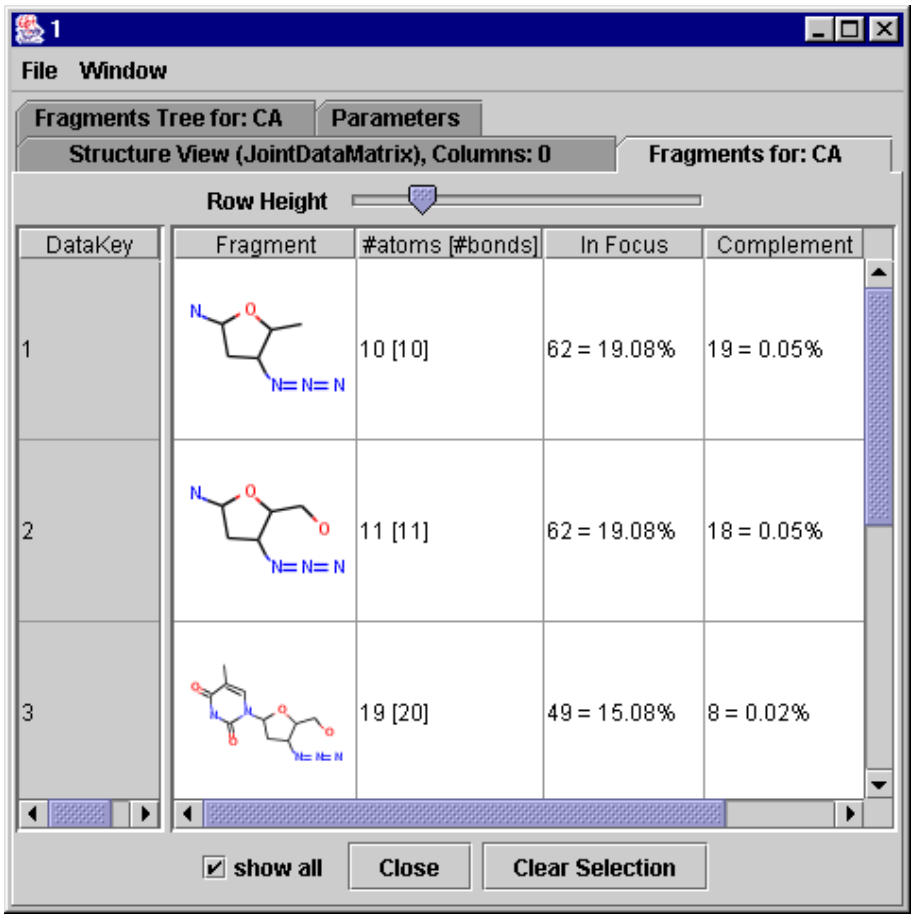


Figure 6.4: The list of discriminative fragments.

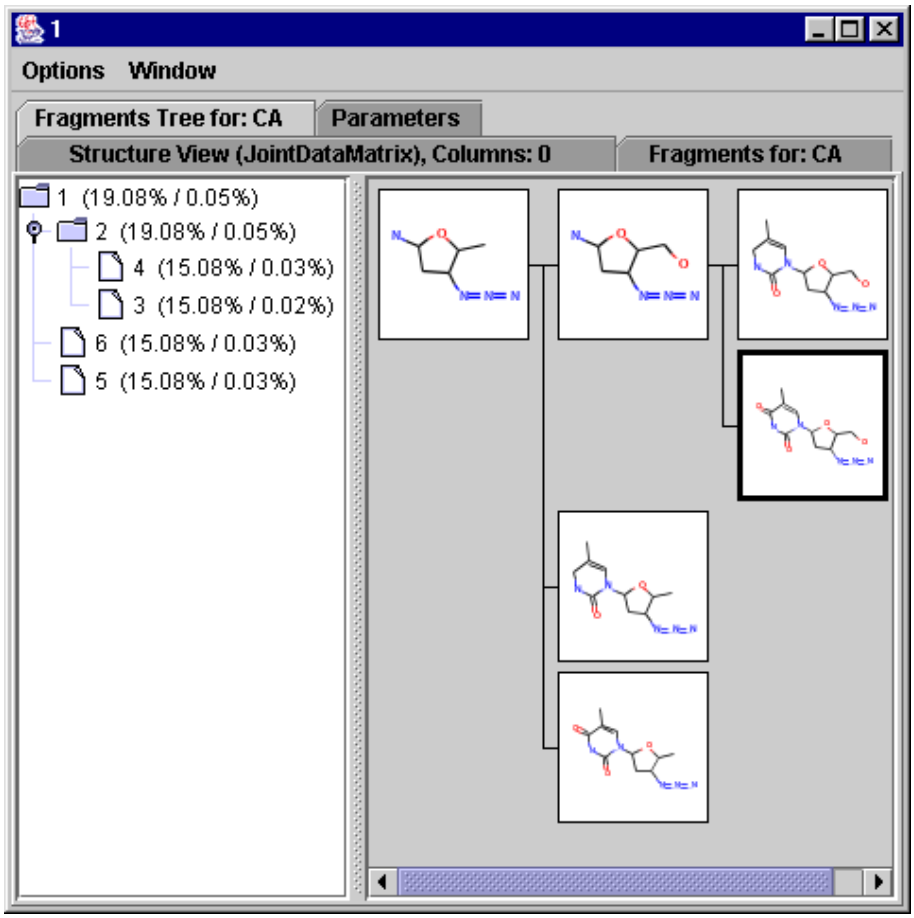


Figure 6.5: The tree of discriminative fragments.

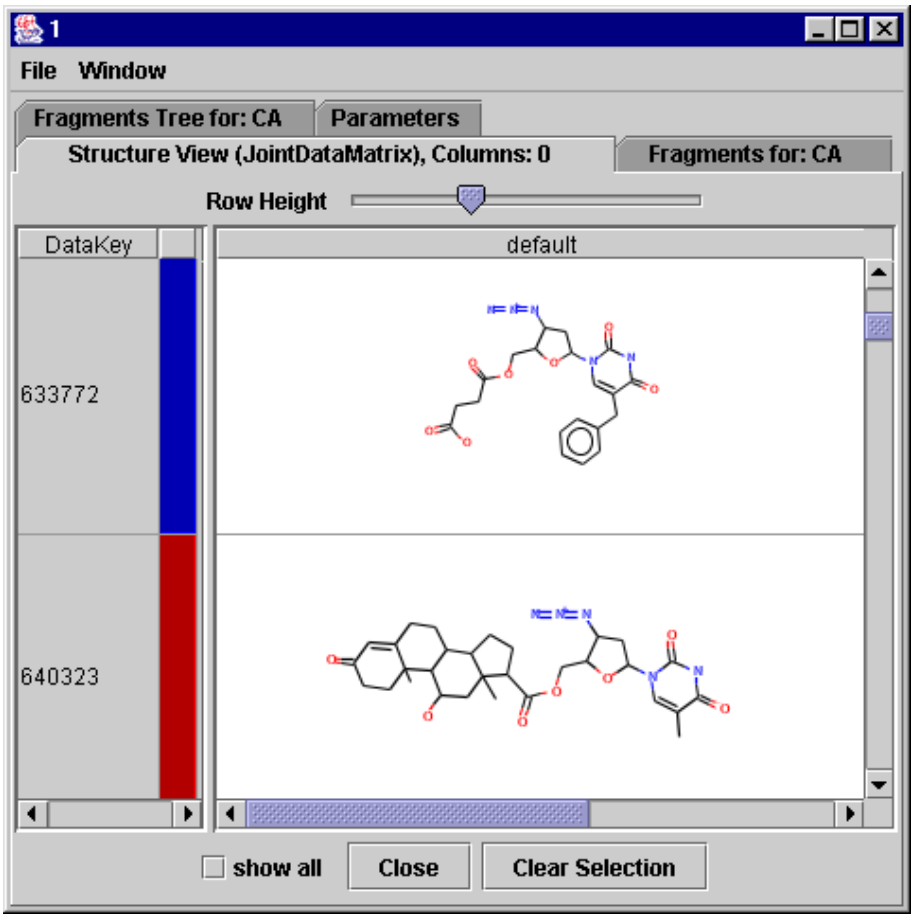


Figure 6.6: The compounds that contain Fragment 3.

7 Conclusions

The context for this diploma thesis is the prediction of biological activity. Borgelt and Berthold (2002) have presented an algorithm that finds discriminative fragments which can be used as classification models for biological activity. This work is the foundation-stone to this thesis.

A graph theoretical description of the algorithm from Borgelt and Berthold is followed by a detailed explanation of their approach with examples. Some improvements are described as well but the main change to the basis algorithm is the treatment of rings as one unit. This leads to a smaller problem and the output are more meaningful fragments at the same time.

Next we take up the work of Kramer et al. (2001) and describe that the use of monotonic and anti-monotonic constraints leads to a solution space that allows to stop the search at the upper border. We have introduced the reduced solution space which contains only those fragments that are necessary to reproduce the fragments of the complete solution space and their frequencies in the database.

The next topic of the thesis is the big area of fuzzy comparisons of molecules. We have shown that fuzzy comparisons of fragments can be viewed as collecting fragments into fuzzy unions. We introduced a modification of the algorithm which allows a certain number of fuzzy atom matches. We believe that the topic of fuzzy matches is worth further studies because an exact match can only solve a part of the problem. A fuzzy approach with highly customizable context-dependent fuzzification would be closer to that what chemists and biologists expect.

Although many parts of this thesis strike the reader as highly theoretical and detached from a practical application the best proof for the value of the contributions presented here is the fact that the resulting tool is already heavily used within Tripos to help improve the success rates of their chemical synthesis plant in Bude, England.

Bibliography

Agrawal, R., Imielienski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the conference on management of data* (pp. 207–216). New York, USA.

Borgelt, C., & Berthold, M. R. (2002). Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings of the IEEE international conference on data mining (ICDM 2002).* Maebashi, Japan.

Diestel, R. (1997). *Graph theory* (1 ed.). New York, USA: Springer-Verlag.

Eckel, B. (2000). *Thinking in java* [available at http://www.mindview.net] (2 ed.). Prentice Hall.

Eckel, B. (2002). *Thinking in patterns* [available at http://www.mindview.net]. unpublished manuscript.

Fortin, S. (1996). *The graph isomorphism problem* (Tech. Rep. No. TR 96-20). Alberta, Canada: Department of computer science, University of Alberta.

Kramer, S., Raedt, L. de, & Helma, C. (2001). Molecular feature mining in HIV data. In *Proceedings of the 7th acm SIGKDD international conference on knowledge discovery and data mining (KDD-2001)* (pp. 136–143). San Francisco, CA, USA.

Kuramochi, M., & Karipys, G. (2002). *An efficient algorithm for discovering frequent subgraphs* (Tech. Rep. No. TR 02-026). Minneapolis, USA: Department of Computer Science/Army HPC Research Center, University of Minnesota.

Mitchel, T. M. (1997). *Machine learning* (1 ed.). New York, USA: McGraw-Hill.

NCI-HIV. *http://dtp.nci.nih.gov/docs/aids/aids_data.html.*

Weislow, O., Kiser, R., Bader, J., Shoemaker, R., & Boyd, M. (1989). Application to high flux screening of synthetic and natural products for aids antiviral activity. *Journal National Cancer Institute*, *81*, 577–586.

Zaki, M. J. (1998). *Scalable data mining for rules.* Unpublished doctoral dissertation, University of Rochester, New York.

Zaki, M. J., Pathasarathy, S., Ogihara, M., & Li, W. (1997). New algorithms for fast discovery of association rules. In *Proceedings of the 3rd international conference on knowledge discovery and data mining (KDD'97, newport beach, ca)* (pp. 283–296). Menlo Park, CA, USA.